DS-GA 3001: Natural Language Understanding with Distributed Representation

Kyunghyun Cho
New York University

October 13, 2015

Note that this lecture note is and will be constantly updated throughout the semester. Please, check for the latest version frequently.

1 Introduction

This lecture is going to be the only one where I discuss some philosophical, meaning nonpractical, arguments, because accoring to Chris Manning and Hinrich Schuetze, "*even practically-minded people have to confront the issue of what prior knowledge to try to build into their model*" [33].

1.1 Route we will *not* take

1.1.1 What is Language?

The very first question we must ask ourselves before starting this course is the question of what natural language is. Of course, the rest of this course does not in any way require us to know what natural language is, but it is a philosophical question I recommend everyone, including myself, to ponder upon once a while.

When I start talking about languages with anyone, there is a single person who never misses to be mentioned, that is Noam Chomsky. His view has greatly influenced the modern linguistics, and although many linguists I have talked to claim that their work and field have long moved on from Chomsky's, I can feel his shadow all over them.

My first encounter with Chomsky was at the classroom of <Automata> from my early undergrad years. I was not the most attentive student back then, and all I can remember is Chomsky's hierarchy and how it has shaped our view on languages, in this context, programming/computer languages. A large part of the course was dedicated to explaining which class of languages emerges given a set of constraints on a set of *generating rules*, or production rules.

For instance, if we are given a set of generating rules that do not depend on the context/meaning of non-terminal symbols (context-free grammar, CFG), we get a context-free language. If we put a bit of constraints to CFG that each generating rule is such

that a non-terminal symbol is replaced by either a terminal symbol, a terminal symbol by a non-terminal symbol or an empty symbol, then we get a regular grammar. Similarly to CFG, we get a regular language from the regular grammar, and the regular language is a subset of the context-free language.

What Chomsky believes is that this kind of approach applies also to human languages, or natural languages. There exists a set of generating rules that *generates* a natural language. But, then, the obvious question to follow is where those generating rules are. Where are they stored? How are they stored? Do we have separate generating rules for different languages?

1.1.2 Language Understanding

Understanding Human Language Those questions are interesting, but out of scope for this course. Those questions are the ones linguists try to answer. Generative linguistics aims at figuring out what those rules are, how they are combined to form a valid sentence, how they are adapted to different languages and so on. We will leave these to linguists and continue on to our journey of *building a machine that understands human languages*.

Natural Language Understanding So, let's put these questions aside and trust Chomsky that we, humans, are specially designed to store those generating rules somewhere in the brain [13, 11]. Or, better yet, let's trust Chomsky that there's a universal grammar *built in* our brain. In other words, let's say we were born with this set of generating rules for natural languages, and while growing, we have adapted this universal grammar toward our native tongue (language variation).

When we decide to speak of something (whatever that is and however implausible that is), our brain quickly picks up a sequence of some of those generating rules and starts generating a sentence accordingly. Of course, those rules do not generate a sentence directly, but generates a sequence of control signals to move our muscles to make sound. When heard by other people who understand your language, the sound becomes a sentence.

In our case, we are more interested in a *machine* hearing that sound, or a sentence from here on. When a machine heard this sentence, what would/should a *language understanding machine* do to understand a language, or more simply a sentence? Again, we are assuming that this sentence was generated from applying a sequence of the existing generating rules.

Under our assumption, a natural first step that comes to my mind is to figure out that sequence of the generating rules which led to the sentence. Once the sequence is found, or in a fancier term, inferred, the next step will be to figure out what kind of mental state of the speaker led to those generating rules.

Let's take an example sentence "*Our company is training workers*" (from Sec. 1.3 of [33]), which is a horrible choice, because this was used as an example of ambiguity in parsing. Regardless, a speaker obviously has an awesome image of her company which trains its workers and wants to tell a machine about this. This mental state is

used to select the following generating rules (assuming a phrase structure grammar)[1]:

```
(ROOT
  (S
      (NP (PRP$ Our) (NN company))
          (VP (VBZ is)
                (VP (VBG training)
                        (NP (NNS workers))))))
```

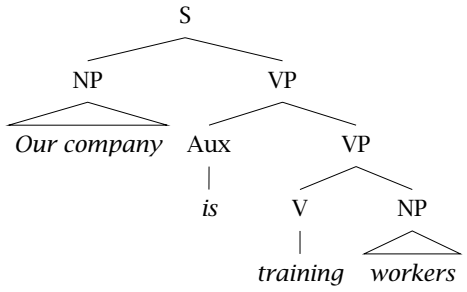

Figure 1: A parse of "*Our company is training workers*"

The machine hears the sentence "*Our company is training workers*" and infers the parse in Fig. 1. Then, we can make a simple set of rules (again!) to let the machine answer questions about this sentence, kinds of questions that imply that the machine has understood the sentence (language). For instance, given a question "*Who is training workers?*", the machine can answer by noticing that the question is asking for the subject of the verb phrase "*is training*" acted on the object "*workers*" and that the subject is "*Our company*".

Side Note: Bayesian Language Understanding This generative view of languages fits quite well with Bayesian modelling (see, e.g., [36].) There exists a hidden mechanism, or a set of generating rules and a rule governing their composition, which can be modelled as a latent variable Z. Given these rules, a language or a sentence X is generated according to the conditional distribution $P(X|Z)$. Then, understanding language (by humans) is equivalent to computing the posterior distribution over all possible sets of generating rules and their compositional rules (i.e., $P(Z|X)$.) This answers the question of what is the most likely mechanism underlying the observed language.

Furthermore, from the perspective of machines, Bayesian approach is attractive. In this case, we assume to know *the* set of rules in advance and let the latent variable Z denote the specific configuration (use) of those rules. Given this sequence of applying the rules, a sentence X is generated via the conditional distribution $P(X|Z)$. Machine understanding of language is equivalent to inferring the posterior distribution over Z given X.

[1] Stanford Parser: `http://nlp.stanford.edu:8080/parser`

For more details about Bayesian approaches (in the context of machine learning), please, refer to [7] or take the course DS-GA 1005 Inference and Representation by Prof. David Sontag.

Understanding vs. Using What's clear from this example is that in this generative view of languages, there is a clear separation between understanding and using. Inferring the generating rules from a given sentence is *understanding*, and answering a question based on this understanding, *using*, is a separate activity. Understanding part is done when the underlying (true) structure has been determined regardless of how this understanding be used.

To put it in a slightly different wording, language understanding does not require its use, or downstream tasks. In this road that we will *not* take in this course, understanding exists as it is, regardless of what the understood insight/knowledge will be used for. And, this is the reason why we do not walk down this road.

1.2 Road we *will* take

1.2.1 Language as a Function

In this course, we will view a natural/human language as "*a system intended to communicate ideas from a speaker to a hearer*" [43]. What this means is that we do not view a language as a separate entity that exists on its own. Rather, we view a whole system or behaviour of *communication* as a language. Furthermore, this view dictates that we must take into account the world surrounding a speaker and a hearer in order to understand language.

Under this view of language, language or rather its usage become somewhat similar to action or behaviour. Speaking of something is equivalent to acting on a listener, as both of them influence the listener in one way or another. The purpose of language is then to influence another by efficiently communicate one's will or intention.[2] This hints at how language came to be (or may have come to be): (evolution) language has evolved to facilitate the exchange of ideas among people (learning) humans learn language by being either encouraged or punished for the use of language. This latter view on how language came to be is similar in spirit to the behaviourism of B. F. Skinner ("*necessary mediation of reinforcement by another organism*" [41].)

This is a radical departure from the generative view of human language, where language existed on its own and its understanding does not necessarily require the existence of the outside world nor the existence of a listener. It is no wonder why Chomsky was so harsh in criticizing Skinner's work in [13]. This departure, as I see it, is the departure toward a functional view of language. *Language is a function of communication*.

[2] Chomsky does not agree: "*it is wrong to think of human use of language as characteristically informative, in fact or in intention.*" [14].

1.2.2 Language Understanding as a Function Approximation

Let's make a large jump here such that we consider this function as a mathematical function. This function (called language) takes as input the state of the surrounding world, the speaker's speech, either written, spoken or signed and the listener's mental state[3] Inside the function, the listener's mental state is updated to incorporate the new idea from the speaker's speech. The function then returns a response by the listener (which may include "no response" as well) and a set of non-verbal action sequences (what would be the action sequence if the speaker insulted the listener?).

In this case, language understanding, both from humans' and machines' perspective, boils down to figuring out the internal working of this function. In other words, we understand language by learning the internal mechanism of the function. Furthermore, this view suggests that the underlying structures of language are heavily dependent on the surrounding environment (context) as well as on the target task. The former (context dependence) is quite clear, as the function takes as input the context, but the latter may be confusing now. Hopefully, this will become clearer later in the course.

How can we approximate this function? How can we figure out the internal working mechanism of this function? What tools do we have?

Language Understanding by Machine Learning This functional view of languages suddenly makes machine learning a very appealing tool for understanding human languages. After all, function approximation is *the* core of machine learning. Classification is a classical example of function approximation, clustering is a function approximation where the target is not given, generative modeling learns a function that returns a probability of an input, and so on.

When we approximate a function in machine learning, the prime ingredient is data. We are given data which was either generated from this function (unsupervised learning) or well fit this function (supervised learning), based on which we adjust our approximation to the function, often iteratively, to best fit the data. But, I must note here that it does not matter how well the approximated function fits the data it was fitted to, but matters how well this approximation fits *unseen* data.[4]

In language understanding, this means that we collect a large data set of input and output pairs (or conversations together with the recording of the surrounding environment) and fit some arbitrary function to well predict the output given an input. We probably want to evaluate this approximation in a novel conversation. If this function makes a conversation just like a person, voilà, we made a machine that passed the Turing test. Simple, right?

Problem Unfortunately, as soon as we try to do this, we run into a big problem. This problem is not from machine learning nor languages, but the definition of this function of language.

Properly approximating this function requires us to either simulate or record the whole world (in fact, the whole universe.) For, this function takes as input and main-

[3] We assume here that a such thing exists however it is represented in our brain.

[4] This is a matter of generalization, and we will talk about this more throughout the course.

tains as internal state the surrounding world (context) and the mental state of the individual (speaker.) This is unavoidable, if we wanted to very well approximate this function as a whole.

It is unclear, however, whether we want to approximate the full function. For a human to survive, yes, it is likely that the full function is needed. But, if our goal is restricted to a certain task (such as translation, language modelling, and so on), we may not want to approximate this function fully. We probably want to approximate only a subset of this whole function. For instance, if our goal is to understand the process of translation from one language to another, we can perhaps ignore all but the speech input to the function and all but the speech output from the function, because often a (trained) person can translate a sentence in one language to another without knowing the whole context.

This latter approach to language understanding–approximating a partial function of languages– will be at the core of this course. We will talk about various language tasks that are a part of this whole function of language. These tasks will include, but are not limited to, language modelling, machine translation, image/video description generation and question answering. For these tasks and potentially more, we will study how to use machine learning, or more specifically deep learning, to solve these tasks by approximating sub-functions of language.

2 Function Approximation as Supervised Learning

Throughout this course, we will extensively use artificial neural networks[5] to approximate (a part of) the function of natural language. This makes it necessary for us to study the basics of neural networks first, and this lecture and a couple of subsequent ones are designed to serve this purpose.

2.1 Function Approximation: Parametric Approach

2.1.1 Expected Cost Function

Let us start by defining a data distribution p_{data}. p_{data} is defined over a pair of input and output vectors, $\mathbf{x} \in \mathbb{I}^d$ and $\mathbf{y} \in \mathbb{O}^k$, respectively. $\mathbb{I}$ and $\mathbb{O}$ are respectively sets of all possible input and output values, such as $\mathbb{R}$, $\{0,1\}$ and $\{0,1,\ldots,L\}$. This data distribution is not known to us.

The goal is to find a relationship between $\mathbf{x}$ and $\mathbf{y}$. More specifically, we are interested in finding a function $f : \mathbb{R}^d \rightarrow \mathbb{O}^k$ that generates the output $\mathbf{y}$ given its corresponding input $\mathbf{x}$. The very first thing we should do is to put some constraints on the function f to make our search for the correct f a bit less impossible. In this lecture, and throughout the course, I will consider only a parametric function f, in which case the function is fully specified with a set of parameters θ.

Next, we must define a way to measure how well the function f approximates the underlying mechanism of generation ($\mathbf{x} \rightarrow \mathbf{y}$). Let's denote by $\hat{\mathbf{y}}$ the output of the

[5] From here on, I will simply drop artificial and call them neural networks. Whenever I say "neural network", it refers to artificial neural networks.

function with a particular set θ of parameters and a given input $\mathbf{x}$:

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x})$$

How well f approximates the true generating function is equivalent to how far $\hat{\mathbf{y}}$ is from the correct output $\mathbf{y}$. Let's use $D(\hat{\mathbf{y}}, \mathbf{y})$ for now call this distance[6] between $\hat{\mathbf{y}}$ and $\mathbf{y}$

It is clear that we want to find θ that minimizes $D(\hat{\mathbf{y}}, \mathbf{y})$ for every pair in the space $(RR^d \times \mathbb{O}^k)$. But, wait, every pair equally likely? Probably not, for we do not care how well f_θ approximates the true function, when a pair of input $\mathbf{x}$ and output $\mathbf{y}$ is unlikely, meaning we do not care how bad the approximation is, if $p_{\text{data}}(\mathbf{x}, \mathbf{y})$ is small. However, this is a bit difficult to take into account, as we must decided on the threshold below which we consider any pair irrelevant.

Hence, we *weight* the distance between the approximated $\hat{\mathbf{y}}$ and the correct $\mathbf{y}$ of each pair $(\mathbf{x}, \mathbf{y})$ in the space by its probability $p(\mathbf{x}, \mathbf{y})$. Mathematically saying, we want to find

$$\arg\min_\theta \int_{\mathbf{x}} \int_{\mathbf{y}} p_{\text{data}}(\mathbf{x}, \mathbf{y}) D(\hat{\mathbf{y}}, \mathbf{y}) \mathrm{d}\mathbf{x} \mathrm{d}\mathbf{y},$$

where the integral $\int$ should be replaced with the summation $\sum$ if any of $\mathbf{x}$ and $\mathbf{y}$ is discrete.

We call this quantity being minimized with respect to the parameters θ a cost function $C(\theta)$. This is equivalent to computing the *expected* distance between the predicted output $\hat{\mathbf{y}}$ and the correct one $\mathbf{y}$:

$$C(\theta) = \int_{\mathbf{x}} \int_{\mathbf{y}} p_{\text{data}}(\mathbf{x}, \mathbf{y}) D(\hat{\mathbf{y}}, \mathbf{y}) \mathrm{d}\mathbf{x} \mathrm{d}\mathbf{y}, \tag{1}$$

$$= \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p_{\text{data}}} [D(\hat{\mathbf{y}}, \mathbf{y})] \tag{2}$$

This is often called an expected loss or risk, and minimizing this cost function is referred to as *expected risk minimization* [42].

Unfortunately $C(\theta)$ cannot be (exactly) computed for a number of reasons. The most important reason among them is simply that we don't know what the data distribution p_{data} is. Even if we have access to p_{data}, we can exactly compute $C(\theta)$ only with heavy assumptions on both the data distribution and the distance function.[7]

2.1.2 Empirical Cost Function

This does not mean that we are doomed from the beginning. Instead of the full-blown description of the data distribution p_{data}, we will assume that someone miraculously gave us a finite set of pairs drawn from the data distribution. We will call this a training set D:

$$D = \left\{ (\mathbf{x}^1, \mathbf{y}^1), \ldots, (\mathbf{x}^N, \mathbf{y}^N) \right\}.$$

[6] Note that we do not require this distance to satisfy the triangular inequality, meaning that it does not have to be a distance. However, I will just call it distance for now.

[7] Why?

As we have access to the samples from the data distribution, we can use Monte Carlo method to approximate the expected cost function $C(\theta)$ such that

$$C(\theta) \approx \tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} D(\hat{\mathbf{y}}^n, \mathbf{y}^n). \tag{3}$$

We call this approximate $\tilde{C}(\theta)$ of the expected cost function, an empirical cost function (or empirical risk or empirical loss.)

Because empirical cost function is readily computable, we will mainly work with the empirical cost function not with the expected cost function. However, keep in mind that at the end of the day, the goal is to find a set of parameters that minimizes the *expected* cost.

2.2 Learning as Optimization

We often call this process of finding a good set of parameters that minimizes the expected cost *learning*. This term is used from the perspective of a machine which implements the function f_θ, as it *learns* to approximate the true generating function f from training data.

From what I have described so far, it may have become clear even without me mentioning that learning is *optimization*. We have a clearly defined function (the empirical cost function $\tilde{C}$) which needs to be minimized with respect to its input θ.

2.2.1 Gradient-based Local Iterative Optimization

There are many optimization algorithms one can use to find a set of parameters that minimizes $\tilde{C}$. Sometimes, you can even find the optimal set of parameters in a closed form equation.[8] In most cases, because there is no known closed-form solution, it is typical to use an iterative optimization algorithm (see [17] for in-depth discussion on optimization.)

By an *iterative* optimization, I mean an algorithm which refines its estimate of the optimal set of parameters little by little until the values of the parameters converge to the optimal (expected) cost function. Also, it is worthwhile to note that most iterative optimization algorithms are *local*, in the sense that they do not require us to evaluate

[8] One such example is a linear regression where

- $f_{\theta=\{\mathbf{W}\}}(\mathbf{x}) = \mathbf{W}\mathbf{x}$
- $D(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2$

In this case, the optimal $\mathbf{W}$ is

$$\mathbf{W} = \mathbf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}, \tag{4}$$

where

$$\mathbf{X} = \left[\mathbf{x}^1; \ldots; \mathbf{x}^N\right], \mathbf{Y} = \left[\mathbf{y}^1; \ldots; \mathbf{y}^N\right].$$

Try it yourself!

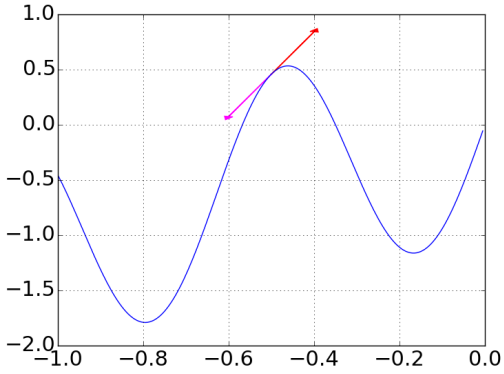

Figure 2: (blue) $f(x) = \sin(10x) + x$. (red) a gradient at $x = -0.6$. (magenta) a negative gradient at $x = -0.6$.

the whole parameter space, but only a small subset along the path from the starting point to the convergence point.[9]

Here I will describe the simplest one among those local iterative optimization algorithms, called gradient descent (GD) algorithm. As the name suggests, this algorithm depends entirely on the gradient of the cost function.[10]

The gradient of a function $\nabla\tilde{C}$ is a vector whose direction points to the direction of the greatest rate of increase in the function's value and whose magnitude measures this rate. At each point θ_t in the parameter space, the gradient of the cost function $\nabla\tilde{C}(\theta_t)$ is the *opposite* direction toward which we want to move the parameters. See Fig. 2 for graphical illustration.

One important point of GD that needs to be mentioned here is on how large a step one takes each time. As clear from the magenta line (the direction opposite to the direction given by the gradient) in Fig. 2, if too large a step is taken toward the negative gradient direction, the optimization process will overshoot and miss the (local) minimum around $x = -0.8$. This step size, or sometimes called learning rate, η is one most important hyperparameter of the GD algorithm.

Now we have all the ingredients for the GD algorithm: $\nabla\tilde{C}$ and η. The GD algorithm iterates the following step:

$$\theta \leftarrow \theta - \eta\nabla\tilde{C}(\theta). \tag{5}$$

The iteration continues until a certain stopping criterion is met, which we will discuss shortly.

2.2.2 Stochastic Gradient Descent

This simple GD algorithm works surprisingly quite well, and it is a fundamental basis upon which many advanced optimization algorithms have been built. I will present a list of few of those advanced algorithms later on and discuss them briefly. But, before

[9] There are *global* optimization algorithms, but they are out of scope for this course. See, for instance, [10] for one such algorithm called Bayesian optimization.

[10] From here on, I will use the cost function to refer to the *empirical* cost function.

going into those advanced algorithms, let's solve one tiny, but significant issue of the GD algorithm.

This tiny, but significant issue arises especially often in machine learning. That is, it is computationally very expensive to compute $\tilde{C}$ and consequently its gradient $\nabla\tilde{C}$, thanks to the ever increasing size of the training set D.

Why is the growing size of the training set making it more and more computationally demanding to compute $\tilde{C}$ and $\nabla\tilde{C}$? This is because both of them are essentially the sum of as many per-sample costs as there are examples in the training set. In other words,

$$\tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\tilde{C}(\mathbf{x}^n, \mathbf{y}^n|\theta),$$
$$\nabla\tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\nabla\tilde{C}(\mathbf{x}^n, \mathbf{y}^n|\theta).$$

And, N goes up to millions or billions very easily these days.

This enormous computational cost involved in each GD step has motivated the *stochastic gradient descent* (SGD) algorithm [38, 8].

First, recall from Eq. (3) that the cost function we minimize is the *empirical* cost function $\tilde{C}$ which is the sample-based approximation to the *expected* cost function C. This approximation was done by assuming that the training examples were drawn randomly from the data distribution p_{data}:

$$C(\theta) \approx \tilde{C}(\theta) = \frac{1}{N}\sum_{n=1}^{N} D(\hat{\mathbf{y}}^n, \mathbf{y}^n).$$

In fact, as long as this assumption on the training set holds, we can always approximate the expected cost function with a fewer number of training examples:

$$C(\theta) \approx \tilde{C}_{\mathscr{M}}(\theta) = \frac{1}{|\mathscr{M}|}\sum_{m\in\mathscr{M}} D(\hat{\mathbf{y}}^m, \mathbf{y}^m),$$

where $M \ll N$ and $\mathscr{M}$ is the indices of the examples in this much smaller subset of the training set. We call this small subset a *minibatch*.

Similarly, this leads to a minibatch-based estimate of the gradient as well:

$$\nabla\tilde{C}_{\mathscr{M}}(\theta) = \frac{1}{|\mathscr{M}|}\sum_{m\in\mathscr{M}} \nabla D(\hat{\mathbf{y}}^m, \mathbf{y}^m).$$

It must now be clear to you where I am headed toward. At each GD step, instead of using the full training set, we will use a small subset $\mathscr{M}$ which is randomly selected to compute the gradient estimate. In other words, we use $\tilde{C}_{\mathscr{M}}$ instead of $\tilde{C}$, and $\nabla\tilde{C}_{\mathscr{M}}$ instead of $\nabla\tilde{C}$, in Eq. (5).

Because computing $\tilde{C}_{\mathscr{M}}$ and $\nabla\tilde{C}_{\mathscr{M}}$ is independent of the size of the training set, we can use SGD to make as many steps as we want without worrying about the growing size of training examples. This is highly beneficial, as regardless of how many training examples you used to compute the gradient, we can only take a tiny step toward

that descending direction. Furthermore, the increased level of noisy in the gradient estimate due to the small sample size has been suspected to help reaching a better solution in high-dimensional non-convex problems (such as those in training deep neural networks) [31].[11]

We can set M to be any constant, and in an extreme, we can set it to 1 as well. In this case, we call it online SGD.[12] Surprisingly, already in 1951, it was shown that using a single example each time is enough for the SGD to converge to a minimum (under certain conditions, obviously) [38].

This SGD algorithm will be at the core of this course and will be discussed further in the future lectures.

2.3 When do we stop learning?

From here on, I assume that we approximate the ground truth function by iteratively refining its set of parameters, in most cases using *stochastic gradient descent*. In other words, learning of a machine that approximates the true generating function f happens gradually as the machine goes over the training examples little by little over time.

Let us go over again what kind of constraints/issue we have first:

1. Lack of access to the expected cost function $C(\theta)$
2. Computationally expensive empirical cost function $\tilde{C}(\theta)$
3. (Potential) non-convexity of the empirical cost function $\tilde{C}(\theta)$

The most severe issue is that we do not have access to the expected cost function which is the one we want to minimize in order to work well with *any* pair of input $\mathbf{x}$ and output $\mathbf{y}$. Instead, we have access to the empirical cost function which is a finite sample approximation to the expected cost function.

Why is this a problem? Because, we do not have a guarantee that the (local) minimum of the empirical cost function corresponds to the (local) minimum of the expected cost function. An example of this mismatch between the expected and empirical cost functions is shown in Fig. 3.

As in the case shown in Fig. 3, it is not desirable to minimize the empirical cost function perfectly. The parameters that perfectly minimize the empirical cost function (in the case of Fig. 3, the slope a of a linear function $f(x) = ax$) will likely be a sub-optimal cost for the expected cost function about which we really care.

2.3.1 Early Stopping

What should we do? There are many ways to avoid this weird contradiction where we want to optimize the cost function well but not too well. Among those, one most important trick is *early stopping*, which is only applicable when iterative optimization is used.

[11] Why would this be the case? It is worth thinking about this issue further.

[12] Okay, this is not true in a strict sense. SGD is an online algorithm with $M = 1$ originally, and using $M > 1$, is a variant of SGD, often called, minibatch SGD. However, as using minibatches ($M > 1$) is almost always the case in practice, I will refer to minibatch SGD as SGD, and to the original SGD as online SGD.

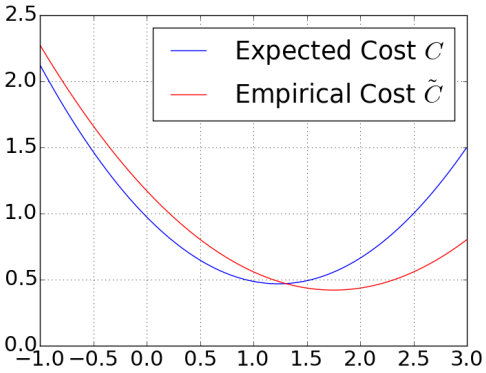

Figure 3: (blue) Expected cost function $C(\theta)$. (red) Empirical cost function $\tilde{C}(\theta)$. The underlying true generating function was $f(x) = \sin(10x) + x$. The cost function uses the squared Euclidean distance. The empirical cost function was computed based on 10 noisy examples of which x's were sampled from the uniform distribution between 0 and 1. For each sample input x, noise from zero-mean Gaussian distribution with standard deviation 0.01 was added to $f(x)$ to emulate the noisy measurement channel.

First, we will split the training set D into two partitions D_{train} and D_{val}.[13] We call them a training set and a validation set, respectively. In practice it is a good idea to keep D much larger than D', because of the reasons that will become clear shortly.

Further, let us define the training cost as

$$\tilde{C}(\theta) = C_{\text{train}}(\theta) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y)\in D_{\text{train}}} D_{\text{train}}(\hat{y}, y), \tag{6}$$

and the validation cost as

$$C_{\text{val}}(\theta) = \frac{1}{|D_{\text{val}}|} \sum_{(x,y)\in D_{\text{val}}} D(\hat{y}, y). \tag{7}$$

With these two cost functions we are all ready to use early stopping now.

After every few updates using SGD (or GD), the validation cost function is evaluated with the current set of parameters. The parameters are updated (i.e., the training cost function is optimized) until the validation cost does not decrease, or starts to increase instead of decreasing.

That's it! It is almost free, as long as the size of the validation set is reasonable, since each evaluation is at most as expensive as computing the gradient of the empirical cost function. Because of the simplicity and effectiveness, this early stopping strategy has become *de facto* standard in deep learning and in general machine learning.

The question that needs to be asked here is what the validation cost function does here. Clearly, it approximates the expected cost function C, similarly to the empirical cost function $\tilde{C}$ as well as the training cost function C_{train}. In the infinite limit of the size of either training or validation set, they should coincide, but in the case of a finite

[13] Later on, we will split it further into three partitions.

set, those two cost functions differ by the noise in sampling (sampling pairs from the data distribution) and observation (noise in $\mathbf{y} = f(\mathbf{x})$.)

The fact that we explicitly optimize the training cost function implies that there is a possibility (in fact, almost surely in practice) that the set of parameters found by this optimization process may capture not only the underlying generating function but also noise in the observation and sampling procedure. This is an issue, because we want our machine to approximate the true generating function not the noise process involved.

The validation cost function measures both the true generating structure as well as noise injected during sampling and observation. However, assuming that noise is not correlated with the underlying generating function, noise introduced in the validation cost function differs from that in the training cost function. In other words, the set of parameters that perfectly minimizes the training cost function (thereby capturing even noise in the training set) will be penalized when measured by the validation cost function.

2.3.2 Model Selection

In fact, the use of the validation cost does not stop at the early stopping. Rather, it has a more general role in model selection. First, we must talk about model selection itself.

This whole procedure of optimization, or learning, can be cast as a process of searching for the best *hypothesis* over the entire space $\mathscr{H}$ of hypotheses. Here, each hypothesis corresponds to each possible function (with a unique set of parameters and a unique functional form) that takes the input $\mathbf{x}$ and output $\mathbf{y}$. In the case of regression ($\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}$), the hypothesis space includes an n-th order polynomial function

$$f(x) = \sum_{\sum_{k=1}^{d} i_k = n, i_k \geq 0} a_{i_1, i_2, \ldots, i_k} \prod_{k'=1}^{d} x_{k'}^{i_k},$$

where $a_{i_1,i_2,\ldots,i_k}$'s are the coefficients, and any other functional form that you can imagine as long as it can process $\mathbf{x}$ and return a real-valued scalar. In the case of neural networks, this space includes all the possible model architectures which are defined by the number of layers, the type of nonlinearities, the number of hidden units in each layer and so on.

Let us use $M \in \mathscr{H}$ to denote one hypothesis.[14] One important thing to remember is that the parameter space is only a subset of the hypothesis space, because the parameter space is defined by a family of hypotheses (the parameter space of a linear function cannot include a set of parameters for a second-order polynomial function.)

Given a definition of expected cost function, we can *score* each hypothesis M by the corresponding cost C_M. Then, the whole goal of function approximation boils down to the search for a hypothesis M with the minimal expected cost function C. But, of course, we do not have access to the expected cost function and resort to the empirical cost function based on a given training set.

The optimization-based approach we discussed so far searches for the best hypothesis based on the empirical cost iteratively. However, because of the issue of *overfitting*

[14] M, because each hypothesis corresponds to one learning *machine*.

which means that the optimization algorithm overshot and missed the local minimum of the expected cost function (because it was aimed at the local minimum of the empirical cost function), I introduced the concept of early stopping based on the validation cost.

This is unfortunately not satisfactory, as we have only searched for the best hypothesis inside a small subset of the whole hypothesis space $\mathscr{H}$. What if another subset of the hypothesis space includes a function that better suits the underlying generating function f? Are we doomed?

It is clearly better to try more than one subsets of the hypothesis space. For instance, for a regression task, we can try linear functions ($\mathscr{H}_1$), quadratic (second-order polynomial) functions ($\mathscr{H}_2$) and sinusoidal functions ($\mathscr{H}_3$). Let's say for each of these subsets, we found the best hypothesis (using iterative optimization and early stopping); $M_{\mathscr{H}_1}$, $M_{\mathscr{H}_2}$ and $M_{\mathscr{H}_3}$. Then, the question is how we should choose one of those hypotheses.

Similar to what we've done with early stopping, we can use the validation cost to compare these hypotheses. Among those three we choose one that has the smallest validation cost $C_{\text{val}}(M)$.

This is one way to do *model selection*, and we will talk about another way to do this later.

2.4 Evaluation

But, wait, if this is an argument for using the validation cost to *early stop* the optimization (or learning), one needs to notice something weird. What is it?

Because we used the validation cost to stop the optimization, there is a chance that the set of parameters we found is optimal for the validation set (whose structure consists of both the true generating function and sampling/observation noise), but not to the general data distribution. This means that we cannot tell whether the function estimate $\hat{f}$ approximating the true generating function f is a good fit by simply early stopping based on the validation cost. Once the optimization is done, we need yet another metric to see how well the learned function estimate $\hat{f}$ approximates f.

Therefore, we need to split the training set not into two partitions but into *three* partitions. We call them a training set D_{train}, a validation set D_{val} and a test set D_{test}. Consequently, we will have three cost functions; a training cost function C_{train}, a validation cost function C_{val} and a test cost function C_{test}, similarly to Eqs. 6–7.

This test cost function is the one we use to compare different hypotheses, or models, fairly. Any hypothesis that worked best in terms of the test cost is the one that you choose.

Let's not Cheat One most important lesson here is that you *must never look at a test set*. As soon as you take a peak at the test set, it will influence your choice in the model structure as well as any other hyperparameters biasing toward a better test cost. The best option is to never ever look at the test set until it is absolutely needed (e.g., need to present your result.)

2.5 Linear Regression for Non-Linear Functions

Let us start with a simple linear function to approximate a true generating function such that

$$\hat{\mathbf{y}} = f(\mathbf{x}) = \mathbf{W}^\top \mathbf{x},$$

where $\mathbf{W} \in \mathbb{R}^{d \times l}$ is the weight matrix. In this case, this weight matrix is the only parameter, i.e., $\theta = \{\mathbf{W}\}$.

The empirical cost function is then

$$\tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{2} \left\| \mathbf{y}^n - \mathbf{W}^\top \mathbf{x}^n \right\|_2^2.$$

The gradient of the empirical cost function is

$$\nabla \tilde{C}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{y}^n - \mathbf{W}^\top \mathbf{x}^n \right)^\top \mathbf{x}^n. \tag{8}$$

With these two well defined, we can use the iterative optimization algorithm, such as GD or SGD, to find the best $\mathbf{W}$ that minimizes the empirical cost function.[15] Or, better is to use a validation set to stop the optimization algorithm at the point of the minimal validation cost function (remember early stopping?)

Now, but we are not too satisfied with a linear network, are we?

2.5.1 Feature Extraction

Why are we not satisfied?

First, we are not sure whether the true generating function f was a linear function. If it is not, can we expect linear regression to approximate the true function well? Of course, not. We will talk about this shortly.

Second, because we were *given* $\mathbf{x}$ (meaning we did not have much control over what we want to measure as $\mathbf{x}$), it is unclear how well $\mathbf{x}$ represents the input. For instance, consider doing a sales forecast of air conditioner at one store which opened five years ago. The input x is the number of days since the opening date of the store (1 Jan 2009), and the output y is the number of units sold on each day.

Clearly, in this example, the relationship between x and y is not linear. Furthermore, perhaps the most important feature for predicting the sales of air conditioners is missing from the input x, which is a month (or a season, if you prefer.) It is likely that the sales bottoms out during the winter (perhaps sometime around December, January and February,) and it hits the peak during summer months (around May, June and July.) In other words, if we look at how far the month is away from July, we can predict the sales quite well even with linear regression.

Let us call this quantity $\phi(x)$, or equivalent *feature*, such that

$$\phi(x) = |m(x) - \alpha|, \tag{9}$$

[15] In fact, looking at Eq. (8), it's quite clear that you can compute the optimal $\mathbf{W}$ analytically. See Eq. (4).

where $m(x) \in \{1, 2, \ldots, 12\}$ is the month of x and $\alpha = 5.5$. With this feature, we can fit linear regression to better approximate the sales figure of air conditioners. Furthermore, we can add yet another feature to improve the predictive performance. For instance, one such feature can be which day of week x is.

This whole process of extracting a good set of features that will make our choice of parametric function family (such as linear regression in this case) is called *feature extraction*. This feature extraction is an important step in machine learning and has often been at the core of many applications such as computer vision (the representative example is SIFT [32].)

Feature extraction often requires heavy knowledge of the domain in which this function approximation is applied. To use linear regression for computer vision, it is a good idea to use computer vision knowledge to extract a good set of features. If we want to use it for environmental problems, we must first notice which features must be important and how they should be represented for linear regression to work.

This is okay for a machine learning practitioner in a particular field, because the person has in-depth knowledge about the field. There are however many cases where there's simply not enough domain knowledge to exploit. To make the matter worse, it is likely that the domain knowledge is not correct, making the whole business of using manually extracted features futile.

3 Neural Networks and Backpropagation Algorithm

3.1 Conditional Distribution Approximation

I have mainly described so far as if the function we approximate or the function we use to approximate returns only a constant value, as in one point $\mathbf{y}$ in the output space. This is however not true, and in fact, the function can return anything including a *distribution* [9, 16, 6].

Let's first decompose the data distribution p_{data} into the product of two terms:

$$p_{\text{data}}(\mathbf{x}, \mathbf{y}) = p_{\text{data}}(\mathbf{x}) p_{\text{data}}(\mathbf{y}|\mathbf{x}).$$

It becomes clear that one way to sample from p_{data} is to sample an input $\mathbf{x}^n$ from $p_{\text{data}}(\mathbf{x})$ and subsequently sample the corresponding output $\mathbf{y}^n$ from the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x}^n)$.

This implies that the function approximation of the generating function ($f : \mathbf{x} \to \mathbf{y}$) is effectively equivalent to approximating the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. This may suddenly sound much more complicated, but it should not alarm you at all. As long as we choose to use a distribution parametrized by a small number of parameters to approximate the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$, this is quite manageable without almost any modification to the expected and empirical cost functions we have discussed.

Let us use $\theta(\mathbf{x})$ to denote a set of parameters for the probability distribution $\tilde{p}(\mathbf{y}|\mathbf{x}, \theta(\mathbf{x}))$ approximating the true, underlying probability distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. As the notation suggests, the function now returns *the parameters of the distribution* $\theta(\mathbf{x})$ given the input $\mathbf{x}$.

For example, let's say $\mathbf{y} \in \{0,1\}^k$ is a binary vector and we chose to use independent Bernoulli distribution to approximate the conditional distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$. In this case, the parameters that define the conditional distribution are the means of k dimensions:

$$\tilde{p}(\mathbf{y}|\mathbf{x}) = \prod_{k'=1}^{k} p(y_{k'}|\mathbf{x}) = \prod_{k'=1}^{k} \mu_{k'}^{y_{k'}} (1-\mu_{k'})^{1-y_{k'}}. \tag{10}$$

Then the function $\theta(\mathbf{x})$ should output a k-dimensional vector of which each element is between 0 and 1.

Another example: let's say $\mathbf{y} \in \mathbb{R}^k$ is a real-valued vector. It is quite natural to use a Gaussian distribution with a diagonal covariance matrix to approximate the conditional distribution $p(\mathbf{y}|\mathbf{x})$:

$$\tilde{p}(\mathbf{y}|\mathbf{x}) = \prod_{k'=1}^{k} \frac{1}{\sqrt{2\pi}\sigma_{k'}} \exp\left(\frac{(y_{k'}-\mu_{k'})^2}{2\sigma_{k'}^2}\right). \tag{11}$$

The parameters for this conditional distribution are $\theta(\mathbf{x}) = \{\mu_1, \mu_2, \ldots, \mu_k, \sigma_1, \sigma_2, \ldots, \sigma_k\}$, where $\mu_k \in \mathbb{R}$ and $\sigma_k \in \mathbb{R}_{>0}$.

In this case of probability approximation, it is natural to use Kullback-Leibler (KL) divergence to measure the distance.[16] The KL divergence from one distribution P to the other Q is defined[17] by

$$\text{KL}(P\|Q) = \int P(\mathbf{x}) \log \frac{P(\mathbf{x})}{Q(\mathbf{x})} \mathrm{d}\mathbf{x}.$$

In our case of function/distribution approximation, we want to minimize the KL divergence from the data distribution $p_{\text{data}}(\mathbf{y}|\mathbf{x})$ to the approximate distribution $\tilde{p}(\mathbf{y}|\mathbf{x})$ averaged over the data distribution $p_{\text{data}}(\mathbf{x})$:

$$C(\theta) = \int p_{\text{data}}(\mathbf{x}) \text{KL}(p_{\text{data}}\|\tilde{p}) \mathrm{d}\mathbf{x} = \int p_{\text{data}}(\mathbf{x}) \int p_{\text{data}}(\mathbf{y}|\mathbf{x}) \log \frac{p_{\text{data}}(\mathbf{y}|\mathbf{x})}{\tilde{p}(\mathbf{y}|\mathbf{x})} \mathrm{d}\mathbf{y}\mathrm{d}\mathbf{x}.$$

But again we do not have access to p_{data} and cannot compute this expected cost function.

Similarly to how we defined the empirical cost function earlier, we must approximate this expected KL divergence using the training set:

$$\tilde{C}(\theta) = \frac{1}{N} \sum_{n=1}^{N} -\log \tilde{p}(\mathbf{y}^n|\mathbf{x}^n). \tag{12}$$

As an example, if we choose to return the binary vector $\mathbf{y}$ as in Eq. (10), the empirical cost function will be

$$\tilde{C}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k'=1}^{k} y_{k'} \log \mu_{k'} + (1-y_{k'}) \log(1-\mu_{k'}),$$

[16] Again, we use a loose definition of the distance where triangular inequality is not enforced.

[17] Why don't I say the KL divergence between two distributions here? Because, the KL divergence is not a symmetric measure, i.e., $\text{KL}(P\|Q) \neq \text{KL}(Q\|P)$.

which is often called a *cross entropy cost*. In the case of Eq. (11),

$$\tilde{C}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k'=1}^{k}\frac{(y_{k'}-\mu_{k'})^2}{2\sigma_{k'}^2} - \log\sigma_{k'}. \tag{13}$$

Do you see something interesting in Eq. (13)? If we assume that the function outputs 1 for all $\sigma_{k'}$'s, we see that this cost function reduces to that using the Euclidean distance between the true output $\mathbf{y}$ and the mean μ. What does this mean?

There will be many occasions later on to discuss more about this perspective when we discuss language modelling. However, one thing we must keep in our mind is that there is nothing different between approximating a function and a distribution.

3.1.1 Why do we want to do this?

Before we move on to the main topic of today's lecture, let's try to understand why we want to output the distribution. Unlike returning a single point in the space, the distribution returned by the function f incorporates both the most likely outcome $\hat{y}$ as well as the uncertainty associated with this value.

In the case of the Gaussian output in Eq. (11), the standard deviation $\sigma_{k'}$, or the variance $\sigma_{k'}^2$, indicates how uncertain the function is about the output centered at $\mu_{k'}$. Similarly, the mean $\mu_{k'}$ of the Bernoulli output in Eq. (10) is directly proportional to the function's confidence in predicting that the k'-th dimension of the output is 1.

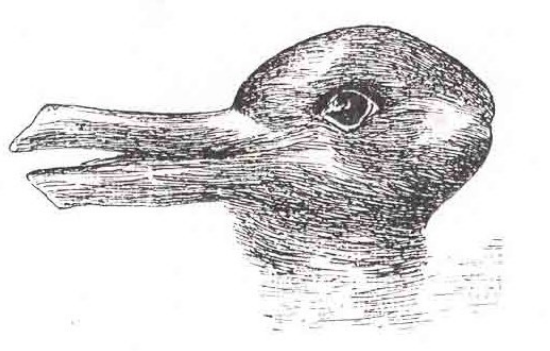

Figure 4: Is this a duck or a rabbit? [28] At the end of the day, we want our function f to return a conditional distribution saying that $p(\text{duck}|\mathbf{x}) = p(\text{rabbit}|\mathbf{x})$, instead of returning *the* answer out of these two possible answers.

This is useful in many aspects, but one important aspect is that it reflects the natural uncertainty of the underlying generating function. One input $\mathbf{x}$ may be interpreted in more than one ways, leading to two possible outputs, which happens more often than not in the real world. For instance, the famous picture in Fig. 4 can be viewed as a picture of a duck or a picture of a rabbit, in which case the function needs to output the probability distribution by which the same probability mass is assigned to both a duck and a rabbit. Furthermore, there is observational noise that cannot easily be identified and ignored by the function, in which case the function should return the uncertainty due to the observational noise along with the most likely (or the average) prediction.

3.1.2 Other Distributions

I have described two distributions (densities) that are widely used:

- Bernoulli distribution: binary classification
- Gaussian distribution: real value regression

Here, let me present one more distribution which we will use almost everyday through this course.

Categorical Distribution: Multi-Class Classification Multi-class classification is a task in which each example belongs to one of K classes. For each input x, the problem reduces to find a probability $p_k(x)$ of the k-th class under the constraint that

$$\sum_{k=1}^{K} p_k(x) = 1$$

It is clear that in this case, the function f returns K values $\{\mu_1, \mu_2, \ldots, \mu_K\}$, each of which is between 0 and 1. Furthermore, the sum of μ_k's must sum to 1. This can be achieved easily by letting f to compute affine transformation of x (or $\phi(x)$) to return K (unbounded) real values followed by a so called *softmax* function [9]:

$$\mu_k = \frac{\exp(w_k^\top \phi(x) + b_k)}{\sum_{k'=1}^{K} \exp(w_{k'}^\top \phi(x) + b_k)}, \tag{14}$$

where $w_k \in \mathbb{R}^{\dim(\phi(x))}$ and $b_k \in \mathbb{R}$ are the parameters of affine transformation.

In this case, the (empirical) cost function based on the KL divergence is

$$C(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \mathbb{I}_{k=y^n} \mu_k, \tag{15}$$

where

$$\mathbb{I}_{k=y^n} = \begin{cases} 1, & \text{if } k = y^n \\ 0, & \text{otherwise} \end{cases}$$

3.2 Feature Extraction is also a Function

We talked about the manual feature extraction in the previous lecture (see Sec. 2.5.1. But, this is quite unsatisfactory, because this whole process of manual feature extraction is heavily dependent on the domain knowledge, meaning that we cannot have a generic principle on which we design features. This raises a question: instead of manually designing features ourselves, is it possible for this to happen automatically?

One thing we notice is that the feature extraction process $\phi(\mathbf{x})$ is nothing but a *function*. A function of a function is a function, right? In other words, we will extend our definition of the function to include the feature extraction function:

$$\hat{\mathbf{y}} = f(\phi(\mathbf{x})).$$

We will assume that the feature extraction function ϕ is also parametrized, and its parameters are included in the set of parameters which includes those of f. As an example, α in Eq. (9) is a parameter of the feature extraction ϕ.

A natural next question is which family of parametric functions we should use for ϕ. We run into the same issue we talked about earlier in Sec. 2.3: the size of hypothesis space is simply too large!

Instead of choosing one great feature extraction function, we can go for a stack of simple transformations which are all learned.[18] Each transformation can be as simple as affine transformation followed by a simple point-wise nonlinearity:

$$\phi_0(\mathbf{x}) = g(\mathbf{W}_0\mathbf{x} + \mathbf{b}_0), \tag{16}$$

where $\mathbf{W}_0$ is the weight matrix, $\mathbf{b}_0$ is the bias and g is a point-wise nonlinearity such as tanh.[19]

One interesting thing is that if the dimensionality of the transformed feature vector $\phi_0(\mathbf{x})$ is *much* larger than that of $\mathbf{x}$, the function $f(\phi_0(\mathbf{x}))$ can approximate any function from $\mathbf{x}$ to $\mathbf{y}$ under some assumptions, even when the parameters $\mathbf{W}_0$ and $\mathbf{b}_0$ are randomly selected! [15]

The problem solved, right? We just put a huge matrix $\mathbf{W}_0$, apply some nonlinear function g to it and fit linear regression as I described earlier. We don't even need to touch $\mathbf{W}_0$ and $\mathbf{b}_0$. All we need to do is replace the input $\mathbf{x}^n$ of all the pairs in the training set to $\phi_0(\mathbf{x}^n)$.

In fact, there is a group of researchers claiming to have figured this out by themselves less than a decade ago (as of 2015) who call this model an *extreme learning machine* [25]. There have been some debates about this so-called extreme learning machine. Here I will not make any comment myself, but would be a good exercise for you to figure out why there has been debates about this.

But, regardlessly, this is not what we want.[20] What we want is to fully tune the whole thing.

3.3 Multilayer Perceptron

The basic idea of multilayer perceptron is to stack a large number of those feature extraction *layers* in Eq. (16) between the input and the output. This idea is as old as the whole field of neural network research, dating back to early 1960s [39]. However, it took many more years for people to figure out a way to tune the whole network, both f and ϕ's together. See [40] and [30], if you are interested in the history.

[18] A great article about this was posted recently in `http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/`.

[19] Some of the widely used nonlinearities are

- Sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$
- Hyperbolic function: $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$
- Rectified linear unit: $\text{rect}(x) = \max(0, x)$

[20] And, more importantly, I will not accept any final project proposal whose main model is based on the ELM.

3.3.1 Example: Binary classification with a single hidden unit

Let us start with the simplest example. The input $x \in \mathbb{R}$ is a real-valued scalar, and the output $y \in \{0,1\}$ is a binary value corresponding to the input's label. The feature extractor ϕ is defined as

$$\phi(x) = \sigma(ux + c), \tag{17}$$

where u and c are the parameters. The function f returns the mean of the Bernoulli conditional distribution $p(y|x)$:

$$\mu = f(x) = \sigma(w\phi(x) + b). \tag{18}$$

In both of these equations, σ is a sigmoid function:

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{19}$$

We use the KL divergence to measure the distance between the true conditional distribution $p(y|x)$ and the predicted conditional distribution $\hat{p}(y|x)$.

$$\begin{aligned}
\mathrm{KL}(p\|\hat{p}) &= \sum_{y\in\{0,1\}} p(y|x) \log \frac{p(y|x)}{\hat{p}(y|x)} \\
&= \sum_{y\in\{0,1\}} p(y|x) \log p(y|x) - p(y|x) \log \hat{p}(y|x).
\end{aligned}$$

Note that the first term in the summation $p(y|x)\log p(y|x)$ can be safely ignored in our case. Why? Because, this does not concern $\tilde{p}$ which is one we change in order to minimize this KL divergence.

Let's approximate this KL divergence with a single sample from $p(y|x)$ and leave only the relevant part. We will call this a per-sample cost:

$$\begin{aligned}
C_x &= -\log \hat{p}(y|x) && (20)\\
&= -\log \mu^y (1-\mu)^{1-y} && (21)\\
&= -y\log\mu - (1-y)\log(1-\mu), && (22)
\end{aligned}$$

where μ is from Eq. (18). It is okay to work with this per-sample cost function instead of the full cost function, because the full cost function is almost always the (unweighted) sum of these per-sample cost functions. See Eq. (3).

We now need to compute the gradient of this cost function C_x with respect to all the parameters w, b, u and c. First, let's start with w:

$$\frac{\partial C_x}{\partial w} = \frac{\partial C_x}{\partial \mu}\frac{\partial \mu}{\partial \underline{\mu}}\frac{\partial \underline{\mu}}{\partial w},$$

which is a simple application of chain rule of derivatives. Compare this to

$$\frac{\partial C_x}{\partial b} = \frac{\partial C_x}{\partial \mu}\frac{\partial \mu}{\partial \underline{\mu}}\frac{\partial \underline{\mu}}{\partial b}.$$

In both equations, $\underline{\mu} = w\phi(x) + b$ which is the input to f.

Both of these derivatives *share* $\frac{\partial C_x}{\partial \mu}\frac{\partial \mu}{\partial \underline{\mu}}$, where

$$\frac{\partial C_x}{\partial \mu}\underbrace{\frac{\partial \mu}{\partial \underline{\mu}}}_{=\mu'} = -\frac{y}{\mu}\mu' + \frac{1-y}{1-\mu}\mu' = \frac{-y+y\mu+\mu-y\mu}{\mu(1-\mu)}\mu' = \frac{\mu-y}{\mu(1-\mu)}\mu' = \mu - y, \quad (23)$$

because the derivative of the sigmoid function $\frac{\partial \mu}{\partial \underline{\mu}}$ is

$$\mu' = \mu(1-\mu).$$

Note that this corresponds to computing the difference between the correct label y and the predicted label (probability) μ.

Given this output derivative $\frac{\partial C_x}{\partial \underline{\mu}}$, all we need to compute are

$$\frac{\partial \underline{\mu}}{\partial w} = \phi(x)$$
$$\frac{\partial \underline{\mu}}{\partial b} = 1.$$

From these computations, we see that

$$\frac{\partial C_x}{\partial w} = (\mu - y)\phi(x), \quad (24)$$
$$\frac{\partial C_x}{\partial b} = (\mu - y). \quad (25)$$

Let us continue on to u and c. We can again rewrite the derivatives w.r.t. these into

$$\frac{\partial C_x}{\partial u} = \frac{\partial C_x}{\partial \underline{\mu}}\frac{\partial \underline{\mu}}{\partial \phi}\frac{\partial \phi}{\partial \underline{\phi}}\frac{\partial \underline{\phi}}{\partial u}$$
$$\frac{\partial C_x}{\partial c} = \frac{\partial C_x}{\partial \underline{\mu}}\frac{\partial \underline{\mu}}{\partial \phi}\frac{\partial \phi}{\partial \underline{\phi}}\frac{\partial \underline{\phi}}{\partial c},$$

where $\underline{\phi}$ is the input to ϕ similarly to $\underline{\mu}$ was to the input to μ.

There are two things to notice here. First, we already have $\frac{\partial C_x}{\partial \underline{\mu}}$ from computing the derivatives w.r.t. w and b, meaning there is no need to re-compute it. Second, $\frac{\partial \underline{\mu}}{\partial \underline{\phi}}$ is shared between the derivatives w.r.t. u and c.

Therefore, we first compute $\frac{\partial \underline{\mu}}{\partial \underline{\phi}}$:

$$\frac{\partial \underline{\mu}}{\partial \phi}\underbrace{\frac{\partial \phi}{\partial \underline{\phi}}}_{=\phi'} = w\phi' = w\phi(x)(1-\phi(x))$$

Next, we compute

$$\frac{\partial \phi}{\partial u} = x$$
$$\frac{\partial \phi}{\partial c} = 1.$$

Now all the ingredients are there:

$$\frac{\partial C_x}{\partial u} = (\mu - y) w \phi(x)(1 - \phi(x)) x$$
$$\frac{\partial C_x}{\partial c} = (\mu - y) w \phi(x)(1 - \phi(x)).$$

The most important lession to learn from here is that most of the computations needed to get the derivatives in this seemingly complicated multilayered computational graph (multilayer perceptron) are *shared*. At the end of the day, the amount of computation needed to compute the gradient of the cost function w.r.t. all the parameters in the network is only as expensive as computing the cost function itself.

3.3.2 Example: Binary classification with more than one hidden units

Let us try to generalize this simple, or rather simplest model, into a slightly more general setting. We will still look at the binary classification but with multiple hidden units and a multidimensional input such that:

$$\phi(x) = Ux + c,$$

where $U \in \mathbb{R}^{l \times d}$ and $c \in \mathbb{R}^l$. Consequently, w will be a l-dimensional vector.

The output derivative $\frac{\partial C_x}{\partial \mu} \frac{\partial \mu}{\partial \underline{\mu}}$ stays same as before. See Eq. (23). However, we note that the derivative of $\underline{\mu}$ with respect to w should now differ, because it's a vector.[21] Let's look at what this means.

The $\underline{\mu}$ can be expressed as

$$\underline{\mu} = w^\top \phi(x) + b = \sum_{i=1}^{l} w_i \phi_i(x) + b. \tag{26}$$

In this case, we can start computing the derivative with respect to each element of w_i separately:

$$\frac{\partial \underline{\mu}}{\partial w_i} = \phi_i(x),$$

and will put them into a vector:

$$\frac{\partial \underline{\mu}}{\partial w} = \left[\frac{\partial \underline{\mu}}{\partial w_1}, \frac{\partial \underline{\mu}}{\partial w_2}, \ldots, \frac{\partial \underline{\mu}}{\partial w_l} \right]^\top = [\phi_1(x), \phi_2(x), \ldots, \phi_l(x)]^\top = \phi(x)$$

[21] The Matrix Cookbook [37] is a good reference for this section.

Then, the derivative of the cost function C_y with respect to w can be written as

$$\frac{\partial C_y}{\partial w} = (\mu - y)\phi(x),$$

in which case nothing really changed from the case of a single hidden unit in Eq. (24).

Now, let's look at $\frac{\partial C_y}{\partial \underline{\phi}}$. Again, because $\phi(x)$ is now a vector, there has to be some changes. Because $\frac{\partial C_y}{\partial \underline{\mu}}$ is already computed, we only need to look at $\frac{\partial \underline{\mu}}{\partial \underline{\phi}}$. In fact, the procedure for computing this is identical to that for computing $\frac{\partial \underline{\mu}}{\partial w}$ due to the symmetry in Eq. (26). That is,

$$\frac{\partial \underline{\mu}}{\partial \underline{\phi}} = w$$

Next, what about $\frac{\partial \underline{\phi}}{\partial \underline{\phi}}$? Because the nonlinear activation function σ is applied element-wise, we can simply compute this derivative for each element in $\phi(x)$ such that

$$\frac{\partial \underline{\phi}}{\partial \underline{\phi}} = \text{diag}\left(\left[\phi_1'(x), \phi_2'(x), \ldots, \phi_l'(x)\right]^\top\right),$$

where diag returns a diagonal matrix of the input vector. In short, we will denote this as ϕ'

Overall so far, we have got

$$\frac{\partial C_y}{\partial \underline{\phi}} = (\mu - y)w^\top \phi'(x) = (\mu - y)(w \odot \text{diag}(\phi'(x))),$$

where $\odot$ is an element-wise multiplication.

Now it is time to compute $\frac{\partial \underline{\phi}}{\partial U}$:

$$\frac{\partial \underline{\phi}}{\partial U} = \frac{\partial U^\top x}{\partial U} = x,$$

according to the Matrix Cookbook [37]. Then, let's look at the whole derivative w.r.t. U:

$$\frac{\partial C_y}{\partial U} = (\mu - y)(w \odot \text{diag}(\phi'(x)))x^\top.$$

Note that all the vectors in this lecture note are *column* vectors.

For c, it's straightforward, since

$$\frac{\partial \underline{\phi}}{\partial c} = 1.$$

3.3.3 Automating Backpropagation

This procedure, presented as two examples, is called a *backpropagation* algorithm. If you read textbooks on neural networks, you see a fancier way to explain this backpropagation algorithm by introducing a lot of fancy terms such as *local error* δ and so on. But, personally I find it much easier to understand backpropagation as a clever application of the chain rule of derivatives to a directed acyclic graph (DAG) in which each node computes a certain function ϕ using the output of the previous nodes. I will refer to this DAG as a computational graph from here on.

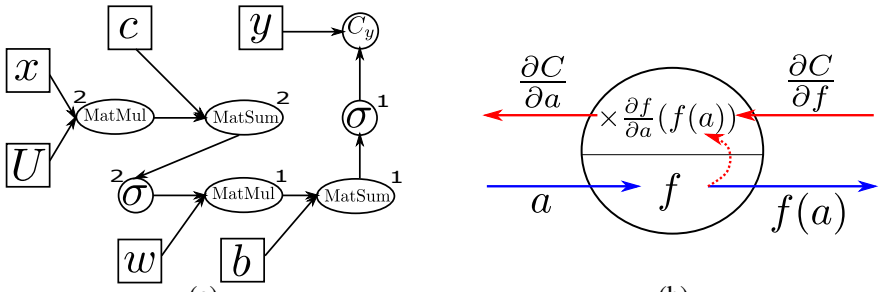

Figure 5: (a) A graphical representation of the computational graph of the example network from Sec. 3.3.2. (b) A graphical illustration of a function node ($\rightarrow$: forward pass, $\leftarrow$: backward pass.)

A typical computational graph looks like the one in Fig. 5 (a). This computational graph has two types of nodes; (1) function node ($\bigcirc$) and (2) variable node ($\square$). There are four different types of function nodes; (1) $\mathrm{MatMul}(A,B) = AB$, (2) $\mathrm{MatSum}(A,B) = A+B$, (3) σ: element-wise sigmoid function and (4) C_y: cost node. The variables nodes correspond to either parameters or data (x and y.) Each function node has a number associated with it to distinguish between the nodes of the same function.

Now, in this computational graph, let us start computing the gradient using the backpropagation algorithm. We start from the last code, C_y, by computing $\frac{\partial C_y}{\partial y}$ and $\frac{\partial C_y}{\partial \sigma^1}$. Then, the function node σ^1 will compute its own derivative $\frac{\partial \sigma^1}{\partial \mathrm{MatSum}^1}$ and multiply it with $\frac{\partial C_y}{\partial \sigma^1}$ passed *back* from the function node C_y. So far we've computed

$$\frac{\partial C_y}{\partial \mathrm{MatSum}^1} = \frac{\partial C_y}{\partial \sigma^1}\frac{\partial \sigma^1}{\partial \mathrm{MatSum}^1} \tag{27}$$

The function node MatSum^1 has two inputs b and the output of MatMul^1. Thus, this node computes two derivatives $\frac{\partial \mathrm{MatSum}^1}{\partial b}$ and $\frac{\partial \mathrm{MatSum}^1}{\partial \mathrm{MatMul}^1}$. Each of these is multiplied with the backpropagated derivative $\frac{\partial C_y}{\partial \mathrm{MatSum}^1}$ from Eq. (27). At this point, we already have the derivative of the cost function C_y w.r.t. one of the parameters b:

$$\frac{\partial C_y}{\partial b} = \frac{\partial C_y}{\partial \mathrm{MatSum}^1}\frac{\partial \mathrm{MatSum}^1}{\partial b}$$

This process continues mechanically until the very beginning of the graph (a set of root variable nodes) is reached. All we need in this process of backpropagating the derivatives is that each function node implements both *forward computation* as well as *backward computation*. In the backward computation, the function node received the derivative from the next function node, evaluates its own derivative with respect to the inputs (at the point of the forward activation) and passes theses derivatives to the corresponding previous nodes. See Fig. 5 (b) for the graphical illustration.

Importantly, the inner mechanism of a function node does not change depending on its context (or equivalently where the node is placed in a computational graph.) In other words, if each type of function nodes is implemented in advance, it becomes trivial to build a complicated neural network (including multilayer perceptrons) and compute the gradient of the cost function (which is one such function node in the graph) with respect to all the parameters as well as all the inputs.

This is a special case, called the reverse mode, of automatic differentiation.[22] It is probably the most valuable tool in deep learning, and fortunately many widely used toolkits such as Theano [5, 1] have implemented this reverse mode of automatic differentiation with an extensive number of function nodes used in deep learning everyday.

Before finishing this discussion on automating backpropagation, I'd like you to think of pushing this even further. For instance, you can think of each function node returning not its numerical derivative on its backward pass, but a computational subgraph computing its derivative. This means that it will return a *computational graph of gradient*, where the output is the derivatives of all the variable nodes (or a subset of them.) Then, we can use the same facility to compute the second-order derivatives, right?

3.3.4 What if a Function is *not* Differentiable?

From the description so far, one thing we notice is that backpropagation works only when each and every function node (in a computational graph) is differentiable. In other words, the nonlinear activation function must be chosen such that almost everywhere it is differentiable. All three activation functions I have presented so far have this property.

Logistic Functions A sigmoid function is defined as

$$\sigma(x) = \frac{1}{1+\exp(-x)},$$

and its derivative is

$$\sigma'(x) = \sigma(x)(1-\sigma(x)).$$

A hyperbolic tangent function is

$$\tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1},$$

[22] If anyone's interested in digging more into the whole field of automatic differentiation, try to Google it and you'll find tons of materials.

and its derivative is

$$\tanh'(x) = \left(\frac{2}{\exp(x) + \exp(-x)} \right)^2.$$

Piece-wise Linear Functions I described a rectified linear unit (rectifier or ReLU, [34, 19]) earlier:

$$\mathrm{rect}(x) = \max(0, x).$$

It is clear that this function is not strictly differentiable, because of the discontinuity at $x = 0$. However, the chance of the input to this rectifier lands exactly at 0 has zero probability, meaning that we can forget about this extremely unlikely event. The derivative of the rectifier in this case is

$$\mathrm{rect}'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

Although the rectifier has become the most widely used nonlinearity, especially, in deep learning's applications to computer vision,[23] there is a small issue with the rectifier. That is, for a half of the input space, the derivative is zero, meaning that the error (the output derivative from Eq. (23)) will be not well propagated through the rectifier function node.

In [20], the rectifier was extended to a maxout unit so as to avoid this issue of the existence of zero-derivative region in the input to the rectifier. The maxout unit of rank k is defined as

$$\mathrm{maxout}(x_1, \ldots, x_k) = \max(x_1, \ldots, x_k),$$

and its derivative as

$$\frac{\partial \mathrm{maxout}}{\partial x_i}(x_1, \ldots, x_k) = \begin{cases} 1, & \text{if } \max(x_1, \ldots, x_k) = x_i \\ 0, & \text{otherwise} \end{cases}$$

This means that the derivative is backpropagated only through one of the k inputs.

Stochastic Variables These activation functions work well with the backpropagation algorithm, because they are differentiable almost everywhere in the input space. However, what happens if a function is non-differentiable at all. One such example is a binary stochastic node, which is computed by

1. Compute $p = \sigma(x)$, where x is the input to the function node.
2. Consider p as a mean of a Bernoulli distribution, i.e., $\mathscr{B}(p)$.
3. Generate one sample $s \in \{0, 1\}$ from the Bernoulli distribution.
4. Output s.

[23] Almost all the winning entries in ImageNet Large Scale Visual Recognition Challenges (ILSVRC) use a convolutional neural network with rectifiers. See `http://image-net.org/challenges/LSVRC/`.

Clearly there is no derivative of this function node.

Does it mean that we're doomed in this case? Fortunately, no. Although I will not discuss about this any further in this course, Bengio et al. [3] provide an extensive list of approaches we can take in order to compute the derivative of the stochastic function nodes.

4 Recurrent Neural Networks and Gated Recurrent Units

After the last lecture I hope that it has become clear how to build a multilayer perceptron. Of course, there are so many details that I did not mention, but are extremely important in practice. For instance, how many layers of simple transformations Eq. (16) should a multilayer perceptron have for a certain task? How wide (equiv. $\dim(\phi_0(\mathbf{x}))$) should each transformation be? What other transformation layers are there? What kind of learning rate η (see Eq. (5)) should we use? How should we schedule this learning rate over training? Answers to many of these questions are unfortunately heavily task-, data- and model-dependent, and I cannot provide any general answer to them.

4.1 Recurrent Neural Networks

Instead, I will move on to describing how we can build a neural network[24] to handle a variable length input. Until now the input x was assumed to be either a scalar or a vector of the fixed number of dimensions. From here on however, we remove this assumption of a fixed size input and consider the case of having a variable length input x.

What do I mean by a *variable length input*? A variable length input x is a *sequence* where each input x has a different number of elements. For instance, the first training example's input x^1 may consist of l^1 elements such that

$$x^1 = (x_1^1, x_2^1, \ldots, x_{l^1}^1).$$

Meanwhile, another example's input x^n may be a sequence of $l^n \neq l^1$ elements:

$$x^n = (x_1^n, x_2^n, \ldots, x_{l^n}^n).$$

Let's go back to very basic about dealing with these kinds of sequences. Furthermore, let us assume that each element x_i is binary, meaning that it is either 0 or 1. What would be the most natural way to write a function that returns the number of 1's in an input sequence $x = (x_1, x_2, \ldots, x_l)$? My answer is to first build a recursive function called ADD1, shown in Alg. 1. This function ADD1 will be called for each element of the input x, as in Alg. 2.

There are two important components in this implementation. First, there is a memory s which counts the number of 1's in the input sequence x. Second, a single function ADD1 is applied to each symbol in the sequence *one at a time* together with the memory s. Thanks to these two properties, our implementation of the function ADD1 can be used with the input sequence of *any length*.

[24] Now, let me begin using a term neural network instead of a general function.

Algorithm 1 A function ADD1

$s \leftarrow 0$
function ADD1(v,s)
  if $v = 0$ **then return** s
  else return $s + 1$
  end if
end function

Algorithm 2 A function ADD1

$s \leftarrow 0$
for $i \leftarrow 1, 2, \ldots, l$ **do** $s \leftarrow \text{ADD1}(x_i, s)$
end for

Now let us generalize this idea of having a memory and a recursive function that works over a variable length sequence. One likely most general case of this idea is a digital computer we use everyday. A computer program is a sequence x of instructions x_i. A central processing unit (CPU) reads each instruction of this program and manipulates its registers according to what the instruction says. Manipulating registers is often equivalent to manipulating any input–output (I/O) device attached to the CPU. Once one instruction is executed, the CPU moves on to the next instruction which will be executed with the content of the registers from the previous step. In other words, these registers work as a memory in this case (s from Alg. 2,) and the execution of an instruction by the CPU corresponds to a recursive function (ADD1 from Alg. 1.)

Both ADD1 and CPU are *hard coded* in the sense that they do what they have been designed and manufactured to do. Clearly, this is not what we want, because nobody knows how to design a CPU or a recursive function for natural language understanding, which is our ultimate goal. Instead what we want is to have a parametric recursive function that is able to read a sequence of (linguistic) symbols and use a memory in order to *understand natural languages*.

To build this parametric recursive function[25] that works on a variable-length input sequence $x = (x_1, x_2, \ldots, x_l)$, we now know that there needs to be a memory. We will use one vector $\mathbf{h} \in \mathbb{R}^{d_h}$ as this memory vector. As is clear from Alg. 1, this recursive function takes as input both one input symbol x_t and the memory vector $\mathbf{h}$, and it returns the updated memory vector. It often helps to *time index* the memory vector as well, such that the input to this function is $\mathbf{h}_{t-1}$ (the memory after processing the previous symbol x_{t-1},) and we use $\mathbf{h}_t$ to denote the memory vector returned by the function. This function is then

$$h_t = f(x_t, \mathbf{h}_{t-1})$$

Now the big question is what kind of parametric form this recursive function f takes? We will follow the simple transformation layer from Eq. (16), in which case we

[25] In neural network research, we call this function a *recurrent neural network.*

get

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1}), \tag{28}$$

where $\phi(x_t)$ is a function that transforms the input symbol (often discrete) into a d-dimensional real-valued vector. $\mathbf{W} \in \mathbb{R}^{d_h \times d}$ and $\mathbf{U}^{d_h \times d_h}$ are parameters of this function. A nonlinear activation function g can be any function, but for now, we will assume that it is an element-wise nonlinear function such as tanh.

4.1.1 Fixed-Size Output y

Because our goal is to approximate an underlying, true function, we now need to think of how we use this recursive function to *return* an output y. As with the case of variable-length sequence input x, y can only be either a fixed-size output, such as a category to which the input x belongs, or a variable-length sequence output. Here let us discuss the case of having a fixed-size output y.

The most natural approach is to use the last memory vector $\mathbf{h}_l$ to produce the output (or more often output distribution.) Consider a task of binary classification where y is either positive (1) or negative (0), in which case a Bernoulli distribution fits perfectly. A Bernoulli distribution is fully characterized by a single parameter μ. Hence,

$$\mu = \sigma(\mathbf{v}^\top \mathbf{h}_l),$$

where $\mathbf{v} \in \mathbb{R}^{d_h}$ is a weight vector, and σ is a sigmoid function.

This now looks very much like the multilayer perceptron from Sec. 3.3. The whole function given an input sequence x computes

$$\mu = \sigma(\mathbf{v}^\top \underbrace{g(\mathbf{W}\phi(x_l) + \mathbf{U}g(\mathbf{W}\phi(x_{l-1}) + \mathbf{U}g(\mathbf{W}\phi(x_{l-2}) + \cdots g(\mathbf{W}\phi(x_1) + \mathbf{U}\mathbf{h}_0)\cdots)))}_{(a)\text{ recurrence}}), \tag{29}$$

where $\mathbf{h}_0$ is an initial memory state which can be simply set to an all-zero vector.

The main difference is that the input is not given only to the first simple transformation layer, but is given to all those transformation layers (one at a time.) Also, each transformation layer *shares* the parameters $\mathbf{W}$ and $\mathbf{U}$.[26] The first two steps of the recurrence part (a) of Eq. (29) are shown as a computational graph in Fig. 6.

As this is not any special computational graph, the whole discussion on how to automate backpropagation (computing the gradient of the cost function w.r.t. the parameters) in Sec. 3.3.3 applies to recurrent neural networks directly, except for one potentially confusing point.

[26] Note that for brevity, I have omitted bias vectors. This should not matter much, as having a bias vector is equivalent to augmenting the input with a constant element whose value is fixed at 1. Why? Because,

$$[\mathbf{W}; \mathbf{b}] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Note that as I have declared before all vectors are *column* vectors.

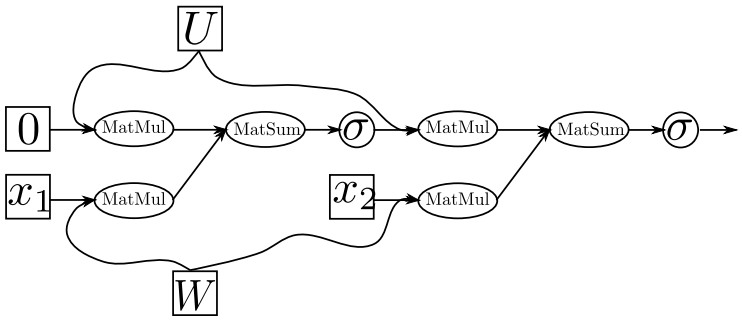

Figure 6: Sample computational graph of the recurrence in Eq. (29).

4.1.2 Multiple Child Nodes and Derivatives

It may be confusing how to handle those parameters that are shared across multiple time steps; **W** and **U** in Fig. 6. In fact, in the earlier section (Sec. 3.3.3), we did not discuss about what to do when the output of one node is fed into multiple function nodes. Mathematically saying, what do we do in the case of

$$c = g(f_1(x), f_2(x), \ldots, f_n(x))?$$

g can be any function, but let us look at two widely used cases:

- Addition: $g(f_1(x), \ldots, f_n(x)) = \sum_{i=1}^{n} f_i(x)$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \frac{\partial f_i}{\partial x}.$$

- Multiplication: $g(f_1(x), \ldots, f_n(x)) = \prod_{i=1}^{n} f_i(x)$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \left(\prod_{j \neq i} f_j(x) \right) \frac{\partial f_i}{\partial x}.$$

From these two cases, we can see that in general

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i \in \{1,2,\ldots,n\}} \frac{\partial g}{\partial f_i} \frac{\partial f_i}{\partial x}.$$

This means that when multiple derivatives are *backpropagated* into a single node, the node should first *sum them* and multiply its summed derivative with its own derivative.

What does this mean for the shared parameters of the recurrent neural network? In

an equation,

$$\begin{aligned}\frac{\partial C}{\partial W} = &\underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatMul}^l} \frac{\partial \mathrm{MatMul}^l}{\partial W} \\ &+ \underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \underbrace{\frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatSum}^{l-1}}}_{(b)} \frac{\partial \mathrm{MatSum}^{l-1}}{\partial \mathrm{MatMul}^{l-1}} \frac{\partial \mathrm{MatMul}^{l-1}}{\partial W} \\ &+ \underbrace{\frac{\partial C}{\partial \mathrm{MatSum}^l}}_{(a)} \underbrace{\frac{\partial \mathrm{MatSum}^l}{\partial \mathrm{MatSum}^{l-1}}}_{(b)} \underbrace{\frac{\partial \mathrm{MatSum}^{l-1}}{\partial \mathrm{MatSum}^{l-2}}}_{(c)} \frac{\partial \mathrm{MatSum}^{l-2}}{\partial \mathrm{MatMul}^{l-2}} \frac{\partial \mathrm{MatMul}^{l-2}}{\partial W} \\ &+ \cdots,\end{aligned} \tag{30}$$

where the superscript l of each function node denotes the layer at which the function node resides.

Similarly to what we've observed in Sec. 3.3.3, many derivatives are shared across the terms inside the summation in Eq. (30). This allows us to compute the derivative of the cost function w.r.t. the parameter W efficiently by simply running the recurrent neural network backward.

4.1.3 Example: Sentiment Analysis

There is a task in natural language processing called *sentiment analysis*. As the name suggests, the goal of this task is to predict the sentiment of a given text. This is definitely one function that a human can do fairly well: when you read a critique's review of a movie, you can easily tell whether the critique likes, hates or is neutral to the movie. Also, even without a star rating of a product on Amazon, you can quite easily tell whether a user like it by reading her/his review of the product.

In this task, an input sequence x is a given text, and the fixed-size output is its label which is almost always one of positive, negative or neutral. Let us assume for now that the input is a *sequence of words*, where each word $\mathbf{x}_i$ is represented as a so-called one-hot vector.[27] In this case, we can use

$$\phi(x_t) = \mathbf{x}_t$$

in Eq. (28).

Once the input sequence, or paragraph in this specific example, is read, we get the last memory state $\mathbf{h}_l$ of the recurrent neural network. We will affine-transform $\mathbf{h}_l$

[27] A one-hot vector is a way to represent a discrete symbol as a binary vector. The one-hot vector $\mathbf{v}_i$ of a symbol $i \in V = \{1, 2, \ldots, |V|\}$ is

$$\mathbf{v}_i = [\underbrace{0, \ldots, 0}_{1,\ldots,i-1}, \underbrace{1}_{i}, \underbrace{0, \ldots, 0}_{i+1,\ldots,|V|}]^\top.$$

followed by the *softmax* function to obtain the conditional distribution of the output $y \in \{1,2,3\}$ (1: positive, 2: neutral and 3: negative):

$$\mu = [\mu_1, \mu_2, \mu_3]^\top = \text{softmax}(\mathbf{V}\mathbf{h}_l),$$

where μ_1, μ_2 and μ_3 are the probabilities of "positive", "neural" and "negative". See Eq. (14) for more details on the softmax function.

Because this network returns a categorial distribution, it is natural to use the (categorical) cross entropy as the cost function. See Eq. (15). A working example of this sentiment analyzer based on recurrent neural networks will be introduced and discussed during the lab session.[28]

4.1.4 Variable-Length Output y: $|x| = |y|$

Let's generalize what we have discussed so far to recurrent neural networks here. Instead of a fixed-size output y, we will assume that the goal is to label each input symbol, resulting in the output sequence $y = (y_1, y_2, \ldots, y_l)$ of the same length as the input sequence x.

What kind of applications can you think of that returns the output sequence as long as the input sequence? One of the most widely studied problems in natural language processing is a problem of classifying each word in a sentence into one of part-of-speech tags, often called POS tagging (see Sec. 3.1 of [33].) Unfortunately, in my personal opinion, this is perhaps the least interesting problem of all time in natural language understanding, but perhaps the most well suited problem for this section.

In its simplest form, we can view this problem of POS tagging as classifying each word in a sentence as one of *noun*, *verb*, *adjective* and *others*. As an example, given the following input sentence x

$$x = (\text{Children}, \text{eat}, \text{sweet}, \text{candy}),$$

the goal is to output

$$y = (\text{noun}, \text{verb}, \text{adjective}, \text{noun}).$$

This task can be solved by a recurrent neural network from the preceding section (Sec. 4.1.1) after a quite trivial modification. Instead of waiting until the end of the sentence to get the last memory state of the recurrent neural network, we will use the *immediate memory state* to predict the label at each time step t.

At each time t, we get the immediate memory state $\mathbf{h}_t$ by

$$\mathbf{h}_t = f(x_t, \mathbf{h}_{t-1}), \tag{31}$$

where f is from Eq. (28). Instead of continuing on to processing the next word, we will first predict the label of the t-th input word x_t.

[28] For those eager to learn more, see `http://deeplearning.net/tutorial/lstm.html` in advance of the lab session.

This can be done by

$$\mu_t = [\mu_{t,1}, \mu_{t,2}, \mu_{t,3}, \mu_{t,4}]^\top = \text{softmax}(\mathbf{V}\mathbf{h}_t).$$

Four $\mu_{t,i}$'s correspond to the probabilities of the four categories; (1) noun, (2) verb, (3) adjective and (4) others.

From this output distribution at time step t, we can define a *per-step*, *per-sample* cost function:

$$C_{x,t}(\theta) = -\sum_{k=1}^{K} \mathbb{I}_{k=y} \mu_{t,k}, \tag{32}$$

where K is the number of categories, four in this case. We discussed earlier in Eq. (15). Naturally a per-sample cost function is defined as the sum of these per-step, per-sample cost functions:

$$C_x(\theta) = -\sum_{t=1}^{l} \sum_{k=1}^{K} \mathbb{I}_{k=y} \mu_{t,k}. \tag{33}$$

Incorporating the Output Structures This formulation of the cost function is equivalent to maximizing the log-probability of the correct output sequence given an input sequence, where the conditional log-probability is defined as

$$\log p(y|x) = \underbrace{\sum_{t=1}^{l} \underbrace{\log p(y_t|x_1, \ldots, x_t)}_{\text{Eq. (32)}}}_{\text{Eq. (33)}}. \tag{34}$$

This means that the network is predicting the label of the t-th input symbol using only the input symbols read up to that point (i.e., $x_1, x_2, \ldots, x_t$.)

In other words, this means that the recurrent neural network is *not* taking into account the structure of the output sequence. For instance, even without looking at the input sequence, in English it is well known that the probability of the next word being a noun increases if the current word is an adjective.[29] This kind of structures in the output are effectively ignored in this formulation.

Why is this so in this formulation? Because, we have made an assumption that the output symbols $y_1, y_2, \ldots, y_l$ are mutually independent conditioned on the input sequence. This is clear from Eq. (34) and the definition of the conditional independence:

$$\begin{aligned} & Y_1 \text{ and } Y_2 \text{ are conditionally independent dependent on } X \\ \iff & p(Y_1, Y_2|X) = p(Y_1|X)p(Y_2|x). \end{aligned}$$

If the underlying, true conditional distribution obeyed this assumption of conditional independence, there is no worry. However, this is a very strong assumption for

[29] Okay, this requires a more thorough analysis, but for the sake of the argument, which does not have to do anything with actual POS tags, let's believe that this is indeed the case.

many of the tasks we run into, apparently from the example of POS tagging. Then, how can we exploit the structure in the output sequence?

One simple way is to make a less strong assumption about the conditional probability of the output sequence y given x. For instance, we can assume that

$$\log p(y|x) = \sum_{i=1}^{l} \log p(y_i|y_{<i}, x_{\leq i}),$$

where $y_{<i}$ and $x_{\leq i}$ denote all the output symbols before the i-th one and all the input symbols up to the i-th one, respectively.

Now the question is how we can incorporate this into the existing formulation of a recurrent neural network from Eq. (31). It turned out that the answer is extremely simple. All we need to do is to compute the memory state of the recurrent neural network based not only on the current input symbol x_t and the previous memory state $\mathbf{h}_{t-1}$, but also on the previous output symbol y_{t-1} such that

$$\mathbf{h}_t = f(x_t, y_{t-1}, \mathbf{h}_{t-1}).$$

Similarly to Eq. (28), we can think of implementing f as

$$f(x_t, y_{t-1}, \mathbf{h}_{t-1}) = g(\mathbf{W}_x \phi_x(x_t) + \mathbf{W}_y \phi_y(y_{t-1}) + \mathbf{W}_h \mathbf{h}_{t-1}).$$

There are two questions naturally arising from this formulation. First, what do we do when computing $\mathbf{h}_1$? This is equivalent to saying what $\phi_y(y_0)$ is. There are two potential answers to this question:

1. Fix $\phi_y(y_0)$ to an all-zero vector
2. Consider $\phi_y(y_0)$ as an additional parameter

In the latter case, $\phi_y(y_0)$ will be estimated together with all the other parameters such as those weight matrices $\mathbf{W}_x$, $\mathbf{W}_y$, $\mathbf{W}_h$ and $\mathbf{V}$.

Inference The second question involves how to handle y_{t-1}. During training, it is quite straightforward, as our cost function (KL-divergence between the underlying, true distribution and the parametric conditional distribution $p(y|x)$, approximated by Monte Carlo method) says that we use the groundtruth value for y_{t-1}'s.

It is however not clear what we should do when we test the trained network, because then we are not given the groundtruth output sequence. This process of finding an output that maximizes the conditional (log-)probability is called *inference*[30]:

$$\hat{y} = \arg\max_{y} \log p(y|x)$$

[30] Okay, I confess. The term *inference* refers to a much larger class of problems, even if we consider only machine learning. However, let me simply use this term to refer to a task of finding the most likely output of a function.

The exact inference is quite straightforward. One can simply evaluate $\log p(y|x)$ for every possible output sequence and choose the one with the highest conditional probability. Unfortunately, this is almost always intractable, as the number of every possible output sequence grows exponentially with respect to the length of the sequence:

$$|\mathscr{Y}| = K^l,$$

where $\mathscr{Y}$, K and l are the set of all possible output sequences, the number of labels and the length of the sequence, respectively. Thus, this is necessary to resort to approximate search over the set $\mathscr{Y}$.

The most naive approach to approximate inference is a greedy one. With the trained model, you predict the first output symbol $\hat{y}_1$ based on the first input symbol x_1 by selecting the category of the highest probability $p(y_1|x_1)$. Now, given $\hat{y}_1$, x_1 and x_2, we compute $p(y_2|x_1,x_2,y_1)$ from which we select the next output symbol $\hat{y}_2$ with the highest probability. We continue this process iteratively until the last output symbol $\hat{y}_l$ is selected.

This is greedy in the sense that any early choice with a high conditional probability may turn out to be unlikely one due to extremely low conditional probabilities later on. It is highly related to the so-called *garden path sentence* problem. To know more about this, read, for instance, Sec. 3.2.4 of [33].

It is possible to alleviate this issue by considering $N < K$ best hypotheses of the output sequence at each time step. This procedure is called *beam search*, and we will discuss more about this in a later lecture on neural machine translation.

4.2 Gated Recurrent Units

4.2.1 Making Simple Recurrent Neural Networks *Realistic*

Let us get back to the analogy we made in Sec. 4.1. We compared a recurrent neural network to how CPU works. Executing a recurrent function f is equivalent to executing one of the instructions on CPU, and the memory state of the recurrent neural network is equivalent to the registers of the CPU. This analogy does sound plausible, except that it is not.

In fact, how a simple recurrent neural network works is far from being similar to how CPU works. I am now talking about how they are implemented in practice, but rather I'm talking at the conceptual level. What is it at the conceptual level that makes the simple recurrent neural network unrealistic?

An important observation we make about the simple recurrent neural network is that it *refreshes* the whole memory state at each time step. This is almost opposite to how the registers on a CPU are maintained. Each time an instruction is executed, the CPU does not clear up the whole registers and repopulate them. Rather, it works only on a small number of registers. All the other registers' values are stored as they were before the execution of the instruction.

Let's try to write this procedure mathematically. Each time, based on the choice of instruction to be executed, a subset of the registers of a CPU, or a subset of the elements in the memory state of a recurrent neural network, is selected. This can be

written down as a binary vector $\mathbf{u} \in \{0,1\}^{n_h}$:

$$u_i = \begin{cases} 0, & \text{if the register's value does not change} \\ 1, & \text{if the register's value will change} \end{cases}$$

With this binary vector, which I will call an *update gate*, a new memory state or a new register value at time t can be computed as a convex interpolation such that

$$\mathbf{h}_t = (1-\mathbf{u}) \odot \mathbf{h}_{t-1} + \mathbf{u} \odot \tilde{\mathbf{h}}_t, \tag{35}$$

where $\odot$ is as usual an element-wise multiplication. $\tilde{\mathbf{h}}_t$ denotes a new memory state or a new register value, after executing the instruction at time t.

Another unrealistic point about the simple recurrent neural network is that each execution considers the whole registers. It is almost impossible to imagine designing an instruction on a CPU that requires to read the values of all the registers. Instead, what almost always happens is that each instruction will consider only a small subset of the registers, which again we can use a binary vector to represent. Let me call it a *reset gate* $\mathbf{r} \in \{0,1\}^{n_h}$:

$$r_i = \begin{cases} 0, & \text{if the register's value will not be used} \\ 1, & \text{if the register's value will be used} \end{cases}$$

This reset gate can be multiplied to the register values *before* being used by the instruction at time t.[31] If we use a recursive function f from Eq. (28), it means that

$$\tilde{\mathbf{h}}_t = f(x_t, \mathbf{r} \odot \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}(\mathbf{r} \odot \mathbf{h}_{t-1})). \tag{36}$$

Now, let us put these two gates that are necessary to make the simple recurrent neural network more realistic into one piece. At each time step, the *candidate* memory state is computed based on a subset of the elements of the previous memory state:

$$\tilde{\mathbf{h}}_t = g(\mathbf{W}\phi(x_t) + \mathbf{U}(\mathbf{r} \odot \mathbf{h}_{t-1}))$$

A new memory state is computed as a linear interpolation between the previous memory state and this candidate memory state using the update gate:

$$\mathbf{h}_t = (1-\mathbf{u}) \odot \mathbf{h}_{t-1} + \mathbf{u} \odot \tilde{\mathbf{h}}_t$$

See Fig. 7 for the graphical illustration.

4.2.2 Gated Recurrent Units

Now here goes a big question: *How are the update* $\mathbf{u}$ *and reset* $\mathbf{r}$ *gates computed?*

If we stick to our analogy to the CPU, those gates must be pre-configured *per instruction*. Those binary gates are dependent on the instruction. Again however, this is not what we want to do in our case. There is no set of predefined instructions, but

[31] It is important to note that this is *not* resetting the actual values of the registers, but only the input to the instruction/recursive function.

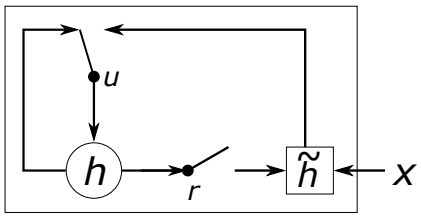

Figure 7: A graphical illustration of a gated recurrent unit [12].

the execution of any instruction corresponds to computing a recurrent *function* based on the input symbol and the memory state from the previous time step (see, e.g., Eq. (28).) Similarly to this what we want with the update and reset gates is that they are computed by a function which depends on the input symbol and the previous memory state.

This sounds like quite straightforward, except that we defined the gates to be binary. This means that whatever the function we use to compute those gates, the function will be a discontinuous function with zero derivative almost everywhere, except at the point where a sharp transition from 0 to 1 happens. We discussed the consequence of having an activation function with zero derivative almost everywhere in Sec. 3.3.4, and the conclusion was that it becomes very difficult to compute the gradient of the cost function efficiently *and* exactly with these discrete activation functions in a computational graph.

One simple solution which turned out to be extremely efficient is to consider those gates not as binary vectors but as real-valued coefficient vectors. In other words, we redefine the update and reset gates to be

$$\mathbf{u} \in [0,1]^{n_h}, \mathbf{r} \in [0,1]^{n_h}.$$

This approach makes these gates *leaky* in the sense that they always allow some leak of information through the gate.

In the case of the reset gate, rather than making a hard decision on which subset of the registers, or the elements of the memory state, will be used, it now decides *how much* information from the previous memory state will be used. The update gate on the other hand now controls how much content in the memory state will be replaced, which is equivalent to saying that it controls how much information will be *kept* from the previous memory state.

Under this definition we can simply use a sigmoid function from Eq. (19) to compute these gates:

$$\begin{aligned}\mathbf{r} &= \sigma(\mathbf{W}_r \phi(x_t) + \mathbf{U}_r \mathbf{h}_{t-1}),\\ \mathbf{u} &= \sigma(\mathbf{W}_u \phi(x_t) + \mathbf{U}_u (\mathbf{r} \odot \mathbf{h}_{t-1})),\end{aligned}$$

where $\mathbf{W}_r$, $\mathbf{U}_r$, $\mathbf{W}_u$ and $\mathbf{U}_u$ are the additional parameters.[32] Since the sigmoid function is differentiable everywhere, we can use the backpropagation algorithm (see Sec. 3.3.3) to compute the derivatives of the cost function with respect to these parameters and estimate them together with all the other parameters.

[32] Note that this is not *the* formulation available for computing the reset and update gates. For instance,

We call this recurrent activation function with the reset and update gates a *gated recurrent unit* (GRU), and a recurrent neural network having this GRU as a gated recurrent network.

Long Short-Term Memory The gated recurrent unit (GRU) is highly motivated by a much earlier work on long short-term memory (LSTM) units [24].[33] The LSTM was proposed in 1997 with the goal of building a recurrent neural network that can learn long-term dependencies across many number of timsteps, which was deemed to be difficult to do so with a simple recurrent neural network.

Unlike the element-wise nonlinearity of the simple recurrent neural network and the gated recurrent unit, the LSTM explicitly separates the memory state $\mathbf{c}_t$ and the output $\mathbf{h}_t$. The output is a small subset of the *hidden* memory state, and only this subset of the memory state is visibly *exposed* to any other part of the whole network.

How does a recurrent neural network with LSTM units decide how much of the memory state it will reveal? As perhaps obvious at this point, the LSTM uses a so-called *output* gate $\mathbf{o}$ to achieve this goal. Similarly to the reset and update gates of the GRU, the output gate is computed by

$$\mathbf{o} = \sigma(\mathbf{W}_o \phi(x_t) + \mathbf{U}_o \mathbf{h}_{t-1}).$$

This output vector is multiplied to the memory state $\mathbf{c}_t$ point-wise to result in the output:

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t).$$

Updating the memory state $\mathbf{c}_t$ closely resembles how it is updated in the GRU (see Eq. (35).) A major difference is that instead of using a single update gate, the LSTM uses two gates, forget and input gates, such that

$$\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{i} \odot \tilde{\mathbf{c}}_t,$$

where $\mathbf{f} \in \mathbb{R}^{n_h}$, $\mathbf{i} \in \mathbb{R}^{n_h}$ and $\tilde{\mathbf{c}}_t$ are the forget gate, input gate and the candidate memory state, respectively.

The roles of those two gates are quite clear from their names. The forget gate decides how much information from the memory state will be *forgotten*, while the input gate controls how much informationa about the new input (consisting of the input symbol and the previous output) will be *inputted* to the memory. They are computed by

$$\begin{aligned} \mathbf{f} &= \sigma(\mathbf{W}_f \phi(x_t) + \mathbf{U}_f \mathbf{h}_{t-1}), \\ \mathbf{i} &= \sigma(\mathbf{W}_i \phi(x_t) + \mathbf{U}_i \mathbf{h}_{t-1}). \end{aligned} \tag{37}$$

---

one can use the following definitions of the reset and update gates:

$$\begin{aligned} \mathbf{r} &= \sigma(\mathbf{W}_r \phi(x_t) + \mathbf{U}_r \mathbf{h}_{t-1}), \\ \mathbf{u} &= \sigma(\mathbf{W}_u \phi(x_t) + \mathbf{U}_u \mathbf{h}_{t-1}), \end{aligned}$$

which is more parallelizable than the original formulation from [12]. This is because there is no more direct dependency between $\mathbf{r}$ and $\mathbf{u}$, which makes it possible to compute them in parallel.

[33] Okay, let me confess here. I was not well aware of long short-term memory when I was designing the gated recurrent unit together with Yoshua Bengio and Caglar Gulcehre in 2014.

The candidate memory state is computed similarly to how it was done with the GRU in Eq. (36):

$$\tilde{\mathbf{c}}_t = g(\mathbf{W}_c\phi(x_t) + \mathbf{U}_c\mathbf{h}_{t-1}),$$

where g is often an element-wise tanh.

All the additional parameters specific to the LSTM– $\mathbf{W}_o, \mathbf{U}_o, \mathbf{W}_f, \mathbf{U}_f, \mathbf{W}_i, \mathbf{U}_i, \mathbf{W}_c$ and $\mathbf{U}_c$– are estimated together with all the other parameters. Again, every function inside the LSTM is differentiable everywhere, and we can use the backpropagation algorithm to efficient compute the gradient of the cost function with respect to all the parameters.

Although I have described one formulation of the long short-term memory unit here, there are many other variants proposed over more than a decade since it was first proposed. For instance, the forget gate in Eq. (37) was not present in the original work [24] but was fixed to 1. Gers et al. [18] proposed the forget gate few years after the LSTM was originally proposed, and it turned out to be one of the most crucial component in the LSTM. For more variants of the LSTM, I suggest you to read [21, 26].[34]

4.3 Why not Rectifiers?

4.3.1 Rectifiers Explode

Let us go back to the *simple recurrent neural network* which uses the simple transformation layer from Eq. (28):

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1}),$$

where g is an element-wise nonlinearity.

One of the most widely used nonlinearities is a hyperbolic tangent function tanh. This is unlike the case in feedforward neural networks (multilayer perceptrons) where a (unbounded) piecewise linear function, such as a rectifier and maxout, has become standard. In the case of feedforward neural networks, you can safely assume that everyone uses some kind of piecewise linear function as an activation function in the network. This has become pretty much standard since Krizhevsky et al. [27] shocked the (computer vision) research community by outperforming all the more traditional computer vision teams in the ImageNet Large Scale Visual Recognition Challenge 2012.[35]

The main difference between logistic functions (tanh and sigmoid function) and piecewise linear functions (rectifiers and maxout) is that the former is bounded from both above and below, while the latter is bounded only from below (or in some cases, not bounded at all [22].[36])

[34] Interestingly, based on the observation in [26], it seems like the plain LSTM with a forget gate and the GRU seem to be close to the optimal gated unit we can find.

[35] `http://image-net.org/challenges/LSVRC/2012/results.html`

[36] A parametric rectifier, or PReLU, is defined as

$$g(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases},$$

where a is a parameter to be estimated together with all the other parameters of a network.

This unbounded nature of piece-wise linear functions makes it difficult for them to be used in recurrent neural networks. Why is this so?

Let us consider the simplest case of unbounded element-wise nonlinearity; a linear function:

$$g(a) = a.$$

The hidden state after l symbols is

$$\begin{aligned}\mathbf{h}_l =& \mathbf{U}(\mathbf{U}(\mathbf{U}(\mathbf{U}(\cdots) + \mathbf{W}\phi(x_{l-3})) + \mathbf{W}\phi(x_{l-2})) + \mathbf{W}\phi(x_{l-1})) + \mathbf{W}\phi(x_l) \\ =& \left(\prod_{l'=1}^{l-1}\mathbf{U}\right)\mathbf{W}\phi(x_1) + \left(\prod_{l'=1}^{l-2}\mathbf{U}\right)\mathbf{W}\phi(x_2) + \cdots + \mathbf{U}\mathbf{W}\phi(x_{l-1}) + \mathbf{W}\phi(x_l), \\ =& \sum_{t=1}^{l} \underbrace{\left(\prod_{l'=1}^{l-t}\mathbf{U}\right)\mathbf{W}\phi(x_t)}_{(a)} \end{aligned} \tag{38}$$

where l is the length of the input sequence.

Let us assume that

- $\mathbf{U}$ is a full rank matrix
- The input sequence is sparse: $\sum_{t=1}^{l}\mathbb{I}_{\phi(x_t)\neq 0} = c$, where $c = O(1)$
- $[\mathbf{W}\phi(x)]_i > 0$ for all i

and consider Eq. (38) (a):

$$\mathbf{h}_l^{t'} = \left(\prod_{l'=1}^{l-t'}\mathbf{U}\right)\mathbf{W}\phi(x_{t'}). \tag{39}$$

Now, let's look at what happens to Eq. (39). First, the eigendecomposition of the matrix $\mathbf{U}$:

$$\mathbf{U} = \mathbf{Q}\mathbf{S}\mathbf{Q}^{-1},$$

where $\mathbf{S}$ is a diagonal matrix whose non-zero entries are eigenvalues. $\mathbf{Q}$ is an orthogonal matrix. Then

$$\prod_{l'=1}^{l-t'}\mathbf{U} = \mathbf{Q}\mathbf{S}^{l-t'}\mathbf{Q}^{-1},$$

and

$$\left(\prod_{l'=1}^{l-t'}\mathbf{U}\right)\mathbf{W}\phi(x_{t'}) = \mathrm{diag}(\mathbf{S}^{l-t'}) \odot (\underbrace{\mathbf{Q}\mathbf{Q}^{-1}}_{=\mathbf{I}}\mathbf{W}\phi(x_{t'})),$$

where $\odot$ is an element-wise product.

What happens if the largest eigenvalue $e_{\max} = \max \operatorname{diag}(\mathbf{S})$ is larger than 1, the norm of $\mathbf{h}_l$ will *explode*, i.e., $\|\mathbf{h}_l\| \to \infty$. Furthermore, due to the assumption that $\mathbf{W}\phi(x_{t'}) > 0$, each element of $\mathbf{h}_l$ will explode to infinity as well. The rate of growth is exponentially with respect to the length of the input sequence, meaning that even when the input sequence is not too long, the norm of the memory state grows quickly if $e_{\max}$ is reasonably larger than 1.

This happens, because the nonlinearity g is *unbounded*. If g is bounded from both above and below, such as the case with tanh, the norm of the memory state is also bounded. In the case of $\tanh : \mathbb{R} \to [-1, 1]$,

$$\|\mathbf{h}_l\| \leq \dim(\mathbf{h}_l).$$

This is one reason why a logistic function, such as tanh and σ, is most widely used with recurrent neural networks, compared to piecewise linear functions.[37] I will call this recurrent neural network with tanh as an element-wise nonlinear function a *simple recurrent neural network*.

4.3.2 Is tanh a Blessing?

Now, the argument in the previous section may sound like tanh and σ are *the* nonlinear functions that one should use. This seems quite convincing for recurrent neural networks, and perhaps so for feedforward neural networks as well, if the network is *deep* enough.

Here let me try to convince you otherwise by looking at how the norm of backpropagated derivative behaves. Again, this is much easier to see if we assume the following:

- $\mathbf{U}$ is a full rank matrix
- The input sequence is sparse: $\sum_{t=1}^{l} \mathbb{I}_{\phi(x_t)\neq 0} = c$, where $c = O(1)$

Similarly to Eq. (38), let us consider a forward computational path until $\mathbf{h}_l$, however without assuming a linear activation function:

$$\mathbf{h}_l = g\left(\mathbf{U}g\left(\mathbf{U}g\left(\mathbf{U}\left(\cdots\right) + \mathbf{W}\phi\left(x_{l-3}\right)\right) + \mathbf{W}\phi\left(x_{l-2}\right)\right) + \mathbf{W}\phi\left(x_{l-1}\right)\right) + \mathbf{W}\phi\left(x_l\right)\right).$$

We will consider a subsequence of this process, in which all the input symbols are 0 except for the first symbol:

$$\mathbf{h}_{l_1} = g\left(\mathbf{U}g\left(\mathbf{U}\left(\cdots g\left(\mathbf{U}\mathbf{h}_{l_0} + \mathbf{W}\phi\left(x_{l_0+1}\right)\right)\right)\right)\right).$$

It should be noted that as l approaches infinity, there will be at least one such subsequence whose length also approaches infinity due to the sparsity of the input we assumed.

From this equation, let's look at

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \phi\left(x_{l_0+1}\right)}.$$

[37] However, it is not to say that piecewise linear functions are never used for recurrent neural networks. See, for instance, [29, 2].

This measures the effect of the (l_0+1)-th input symbol x_{l_0+1} on the l_1-th memory state of the simple recurrent neural network. This is also the crucial derivative that needs to be computed in order to compute the gradient of the cost function using the automated backpropagation procedure described in Sec. 3.3.3.

This derivative can be rewritten as

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \phi\left(x_{l_0+1}\right)} = \underbrace{\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}}}_{(a)} \frac{\partial \mathbf{h}_{l_0+1}}{\partial \underline{\mathbf{h}}_{l_0+1}} \frac{\partial \underline{\mathbf{h}}_{l_0+1}}{\partial \phi\left(x_{l_0+1}\right)}.$$

Among these three terms in the left hand side, we will focus on the first one (a) which can be further expanded as

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} = \left(\underbrace{\frac{\partial \mathbf{h}_{l_1}}{\partial \underline{\mathbf{h}}_{l_1}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_1}}{\partial \mathbf{h}_{l_1-1}}}_{(c)} \right) \left(\underbrace{\frac{\partial \mathbf{h}_{l_1-1}}{\partial \underline{\mathbf{h}}_{l_1-1}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_1-1}}{\partial \mathbf{h}_{l_1-2}}}_{(c)} \right) \cdots \left(\underbrace{\frac{\partial \mathbf{h}_{l_0+2}}{\partial \underline{\mathbf{h}}_{l_0+2}}}_{(b)} \underbrace{\frac{\partial \underline{\mathbf{h}}_{l_0+2}}{\partial \mathbf{h}_{l_0+1}}}_{(c)} \right). \tag{40}$$

Because this is a recurrent neural network, we can see that the analytical forms for the terms grouped by the parentheses in the above equation are identical except for the subscripts indicating the time index. In other words, we can simply only on one of those groups, and the resulting analytical form will be generally applicable to all the other groups.

First, we look at Eq. (40) (b), which is nothing but a derivative of a nonlinear activation function used in this simple recurrent neural network. The derivatives of the widely used logistic functions are

$$\begin{aligned} \sigma'(x) =& \sigma(x)(1-\sigma(x)), \\ \tanh'(x) =& 1-\tanh^2(x), \end{aligned}$$

as described earlier in Sec. 3.3.4. Both of these functions' derivatives are *bounded*:

$$0 < \sigma'(x) \leq 0.25, \tag{41}$$

$$0 < \tanh'(x) \leq 1. \tag{42}$$

In the simplest case in which g is a linear function (i.e., $x = g(x)$,) we do not even need to look at $\left\| \frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t} \right\|$. We simply ignore all the $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}$ from Eq. (40).

Next, consider Eq. (40) (c). In this case of simple recurrent neural network, we notice that we have already learned how to compute this derivative earlier in Sec. 3.3.2:

$$\frac{\partial \underline{\mathbf{h}}_{t+1}}{\partial \mathbf{h}_t} = \mathbf{U}$$

From these two, we get

$$\begin{aligned} \frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} &= \left(\frac{\partial \mathbf{h}_{l_1}}{\partial \underline{\mathbf{h}}_{l_1}} \mathbf{U} \right) \left(\frac{\partial \mathbf{h}_{l_1-1}}{\partial \underline{\mathbf{h}}_{l_1-1}} \mathbf{U} \right) \cdots \left(\frac{\partial \mathbf{h}_{l_0+2}}{\partial \underline{\mathbf{h}}_{l_0+2}} \mathbf{U} \right) \\ &= \prod_{t=l_0+2}^{l_1} \left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t} \mathbf{U} \right). \end{aligned}$$

Do you see how similar it looks like Eq. (39)? If the recurrent activation function f is linear, this whole term reduces to

$$\frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} = \mathbf{U}^{l_1-l_0+1},$$

which according to Sec. 4.3.1, will explode as $l \to \infty$ if

$$e_{\max} > 1,$$

where $e_{\max}$ is the largest eigenvalue of $\mathbf{U}$. When $e_{\max} < 1$, it will *vanish*, i.e., $\| \frac{\partial \mathbf{h}_{l_1}}{\partial \mathbf{h}_{l_0+1}} \| \to 0$, exponentially fast.

What if the recurrent activation function f is *not* linear at all? Let's look at $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U}$ as

$$\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U} = \underbrace{\begin{bmatrix} f_1' & 0 & \cdots & 0 \\ 0 & f_2' & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & f_{n_h}' \end{bmatrix}}_{=\mathrm{diag}\left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\right)} \left(\mathbf{QSQ}^{-1}\right),$$

where we used the eigendecomposition of $\mathbf{U} = \mathbf{QSQ}^{-1}$. This can be re-written into

$$\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U} = \mathbf{Q}\left(\mathrm{diag}\left(\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\right) \odot \mathbf{S}\right)\mathbf{Q}^{-1}.$$

This means that the eigenvalue of $\mathbf{U}$ will be scaled by the derivative of the recurrent activation function at each timestep. In this case, we can bound the maximum eigenvalue of $\frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}\mathbf{U}$ by

$$e_{\max}^t \leq \lambda e_{\max},$$

where λ is the upperbound on $g' = \frac{\partial \mathbf{h}_t}{\partial \underline{\mathbf{h}}_t}$. See Eqs. (41)–(42) for the upperbounds of the sigmoid and hyperbolic tangent functions.

In other words, if the largest eigenvalue of $\mathbf{U}$ is larger than $\frac{1}{\lambda}$, it is *likely* that this temporal derivative of $\mathbf{h}_{l_1}$ with respect to $\mathbf{h}_{l_0+1}$ will *explode*, meaning that its norm will grow exponentially large. In the opposite case of $e_{\max} < \frac{1}{\lambda}$, the norm of the temporal derivative likely shrinks toward 0. The former case is referred to as *exploding gradient*, and the latter *vanishing gradient*. These cases were studied already at the very early years of research in recurrent neural networks [4, 23].

Using tanh is a blessing in recurrent neural networks when running the network forward, as I described in the previous section. This is however not necessarily true in the case of backpropagating derivaties. Especially because, there is a higher chance of vanishing gradient with tanh, or even worse with σ. Why? Because $\frac{1}{\lambda} > 1$ for almost everywhere.

4.3.3 Are We Doomed?

Exploding Gradient Fortunately it turned out that the phenomenon of exploding gradient is quite easy to address. First, it is straightforward to detect whether the exploding gradient happened by inspecting the norm of the gradient fo the cost with respect to the parameters $\left\|\nabla_\theta \tilde{C}\right\|$. If the gradient's norm is larger than some predefined threhold $\tau > 0$, we can simply renormalize the norm of the gradient to be τ. Otherwise, we leave it as it is.

In mathematics,

$$\tilde{\nabla} = \begin{cases} \tau \frac{\nabla}{\|\nabla\|}, & \text{if } \|\nabla\| > \tau \\ \nabla, & \text{otherwise} \end{cases},$$

where we used the shorthand notiation ∇ for $\nabla_\theta \tilde{C}$. $\tilde{\nabla}$ is a rescaled gradient update direction which will be used by the stochastic gradient descent (SGD) algorithm from Sec. 2.2.2. This algorithm is referred to as *gradient clipping* [35].

Vanishing Gradient What about vanishing gradient? But, first, what does vanishing gradient mean? We need to understand the meaning of this phenomenon in order to tell whether this is a problem at all from the beginning.

Let us consider a case the variable-length output where $|x| = |y|$ from Sec. 4.1.4. Let's assume that there exists a clear dependency between the output label y_t and the input symbol $x_{t'}$, where $t' \ll t$. This means that the empirical cost will decrease when the weights are adjusted such that

$$\log p(y_t = y_t^* | \ldots, \phi(x_{t'}), \ldots)$$

is maximized, where y_t^* is the ground truth output label at time t. The value of $\phi(x_{t'})$ has great influence on the t-th output y_t, and the influence can be measured by

$$\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}.$$

Instead of exactly computing $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$, we can approximate it by the finite difference method. Let $\varepsilon \in \mathbb{R}^{\dim(\phi(x_{t'}))}$ be a vector of which each element is a very small real value ($\varepsilon \approx 0$.) Then,

$$\begin{aligned} \frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})} = \lim_{\varepsilon \to 0} (&\log p(y_t = y_t^* | \ldots, \phi(x_{t'}) + \varepsilon, \ldots) \\ &- \log p(y_t = y_t^* | \ldots, \phi(x_{t'}), \ldots,)) \oslash \varepsilon, \end{aligned}$$

where $\oslash$ is an element-wise division. This shows that $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$ computes the difference in the t-th output probability with respect to the change in the value of the t'-th input.

In other words $\frac{\partial \log p(y_t = y_t^* | \ldots)}{\partial \phi(x_{t'})}$ directly reflects the degree to which the t-th output y_t *depends* on the t'-th input $x_{t'}$, *according to the network*. To put it in another way,

$\frac{\partial \log p(y_t=y_t^*|...)}{\partial \phi(x_{t'})}$ reflects how much dependency the *recurrent neural network has captured the dependency between y_t and $x_{t'}$.*

Let's rewrite

$$\frac{\partial \log p(y_t = y_t^*|\dots)}{\partial \phi(x_{t'})} = \frac{\partial \log p(y_t = y_t^*|\dots)}{\partial \mathbf{h}_t} \underbrace{\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \cdots \frac{\partial \mathbf{h}_{t'+1}}{\partial \mathbf{h}_{t'}}}_{(a)} \frac{\partial \mathbf{h}_{t'}}{\partial \phi(x_t)}.$$

The terms marked with (a) looks exactly identical to Eq. (40). We have already seen that this term can easily *vanish* toward zero with a high probability (see Sec. 4.3.2.)

This means that the recurrent neural network is unlikely to capture this dependency. This is especially true when the (temporal) distance between the output and input, i.e., $|t-t'| \gg 0$.

The biggest issue with this vanishing behaviour is that there is no straightforward way to avoid it. We cannot tell whether $\frac{\partial \log p(y_t=y_t^*|...)}{\partial \phi(x_{t'})} \approx 0$ is due to the lack of this dependency in the true, underlying function or due to the wrong configuration (parameter setting) of the recurrent neural network. If we are certain that there are indeed these long-term dependencies, we may simultaneously minimize the following auxiliary term together with the cost function:

$$\sum_{t=1}^{T} \left(1 - \frac{\left\| \frac{\partial \tilde{C}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right\|}{\left\| \frac{\partial \tilde{C}}{\partial \mathbf{h}_{t+1}} \right\|} \right)^2.$$

This term, which was introduced in [35], is minimized when the norm of the derivative does not change as it is being backpropagated, effectively forcing the gradient *not* to vanish.

This term however was found to help significantly only when the target task, or the underlying function, does indeed exhibit long-term dependencies. How can we know in advance? Pascanu et al. [35] showed this with the well-known toy tasks which were specifically designed to exhibit long-term dependencies [23].

4.3.4 Gated Recurrent Units Address Vanishing Gradient

Will the same problems of vanishing gradient happen with the gated recurrent units (GRU) or the long short-term memory units (LSTM)? Let us write the memory state at time t:

$$\begin{aligned}\mathbf{h}_t =& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \left(\mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_{t-1}) \odot \left(\mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + (1-\mathbf{u}_{t-2}) \odot (\cdots) \right) \right) \\ =& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_t) \odot (1-\mathbf{u}_{t-1}) \odot \mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + \cdots\end{aligned}$$

Let's be more specific and see what happens to this with respect to $x_{t'}$:

$$\begin{aligned}\mathbf{h}_t =& \mathbf{u}_t \odot \tilde{\mathbf{h}}_t + (1-\mathbf{u}_t) \odot \mathbf{u}_{t-1} \odot \tilde{\mathbf{h}}_{t-1} + (1-\mathbf{u}_t) \odot (1-\mathbf{u}_{t-1}) \odot \mathbf{u}_{t-2} \odot \tilde{\mathbf{h}}_{t-2} + \cdots \\ &+ \underbrace{\left(\prod_{k=t,\dots,t'+1} (1-\mathbf{u}_k) \right) \odot \mathbf{u}_{t'}}_{(a)} \odot \tanh\left(\mathbf{W}\phi(x_{t'}) + \mathbf{U}\left(\mathbf{r}_{t'} \odot \mathbf{h}_{t'-1} \right) \right), \end{aligned} \tag{43}$$

where $\prod$ is for element-wise multiplication.

What this implies is that the GRU effectively introduces a *shortcut* from time t' to t. The change in $x_{t'}$ will directly influence the value of $\mathbf{h}_t$, and subsequently the t-th output symbol y_t. In other words, all the issue with the simple recurrent neural network we discussed earlier in Sec. 4.3.3.

The update gate controls the strength of these shortcuts. Let's assume for now that the update gate is fixed to some predefined value between 0 and 1. This effectively makes the GRU a leaky integration unit [2]. However, as it is perhaps clear from Eq. (43) that we will inevitably run into an issue. Why is this so?

Let's say we are sure that there are many long-term dependencies in the data. It is natural to choose a large coefficient for the leaky integration unit, meaning the update gate is close to 1. This will definitely help carrying the dependency across many time steps, but this inevitably carries *unnecessary* information as well. This means that much of the representational power of the output function $g_{\text{out}}(\mathbf{h}_t)$ is wasted in *ignoring* those unnecessary information.

If the update gate is fixed to something substantially smaller than 1, all the shortcuts (see Eq. (43) (a)) will exponentially vanish. Why? Because it is a repeated multiplication of a scalar small than 1. In other words, it does not really help to have a leaky integration unit in the place of a simple tanh unit.

This is however not the case with the actual GRU or LSTM, because those update gates are *not* fixed but are adaptive with respect to the input. If the network detects that there is an important dependency being captured, the update gate will be closed ($u_j \approx 0$.) This will effectively strengthen the shortcut connection (see Eq. (43) (a).) When the network detects that there is no dependency anymore, it will open the update gate ($u_j \approx 1$), which effectively cuts off the shortcut. How does the network know, or detect, the existence or lack of these dependencies? Do we need to manually code this up? I will leave these questions for you to figure out.

5 Neural Language Models

Soon to appear

References

[1] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[2] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[3] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[4] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[6] C. M. Bishop. Mixture density networks. 1994.

[7] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[8] L. Bottou. Online algorithms and stochastic approximations. In D. Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[9] J. S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990.

[10] E. Brochu, V. M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv:*`1012.2599 [cs.LG]`, Dec. 2010.

[11] A. Carnie. *Syntax: A generative introduction*. John Wiley & Sons, 2013.

[12] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, Oct. 2014.

[13] N. Chomsky. A review of B. F. skinner's verbal behavior. *Language*, 35(1):26–58, 1959.

[14] N. Chomsky. Linguistic contributions to the study of mind (future). *Language and thinking*, pages 323–364, 1968.

[15] T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326–334, 1965.

[16] J. Denker and Y. Lecun. Transforming neural-net output levels to probability distributions. In *Advances in Neural Information Processing Systems 3*. Citeseer, 1991.

[17] R. Fletcher. *Practical Methods of Optimization*. Wiley-Interscience, New York, NY, USA, 2nd edition, 1987.

[18] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[19] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[20] I. Goodfellow, D. Warde-farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1319–1327, 2013.

[21] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[22] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[23] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*, volume 1. A field guide to dynamical recurrent neural networks. IEEE Press, 2001.

[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1—3):489–501, 2006.

[26] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350, 2015.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[28] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[29] Q. V. Le, N. Jaitly, and G. E. Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[30] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[31] Y. LeCun, L. Bottou, G. Orr, and K. R. Müller. Efficient BackProp. In G. Orr and K. Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 5–50. Springer Verlag, 1998.

[32] D. G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[33] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[34] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[35] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318, 2013.

[36] A. Perfors, J. Tenenbaum, and T. Regier. Poverty of the stimulus? a rational approach. In *Annual Conference*, 2006.

[37] K. B. Petersen, M. S. Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7:15, 2008.

[38] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[39] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[40] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[41] B. F. Skinner. *Verbal behavior*. BF Skinner Foundation, 2014.

[42] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[43] T. Winograd. Understanding natural language. *Cognitive psychology*, 3(1):1–191, 1972.